Unified system of code transformation and execution for heterogeneous multi-core architectures.

Pei Li

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

École Doctorale de Mathématiques et Informatique
SPÉCIALITÉ: Informatique

Par Pei Li

Système unifié de transformation de code et d'exécution pour un passage aux architectures multi-coeurs hétérogènes

Sous la direction de : Raymond Namyst
Encadrante de thèse : Elisabeth Brunet

Soutenue le 17 Décembre 2015

Membres du jury :

M. ROMAN Jean	Professeur	Institut Polytechnique de Bordeaux	Président
M. MEHAUT Jean-François	Professeur	Université de Grenoble 1	Rapporteur
M. SENS Pierre	Professeur	Université Pierre et Marie Curie	Rapporteur
M. CARRIBAULT Patrick	Chercheur	CEA	Examinateur
M. NAMYST Raymond	Professeur	Université de Bordeaux 1	Directeur de thèse
Mme. BRUNET Elisabeth	Maître de conférence	Institut Télécom SudParis	Encadrante de thèse

Remerciement

Cette thèse est le fruit de travail qui appartient non seulement à moi, mais aussi à tous les personnes qui m'a supporté mon travail et ma vie pendant ces 3 ans de thèse. Je profite de cette occasion ici pour exprimer ma sincère gratitude.

Je tiens à remercier en tout premier lieu mon encadrant Elisabeth Brunet. Je vous remercie de m'avoir proposé ce sujet de thèse, m'avoir fait confiance et m'avoir accueilli au sein de votre équipe. Grace à vous, j'ai pu entrer dans un nouveau monde où je n'avais jamais exploré. Quand j'ai rencontré des problèmes sur mes recherches, vous étiez toujours la première personne qui m'a encouragé et m'a donné la suggestion. Je remercie également mon directeur de thèse Raymond Namyst. Vous m'avez guidé la direction de recherche pendant ma thèse. Sans votre conseil, tous ce travail n'aurait pas été possible! Je vous remercie aussi de m'avoir accueilli au sein de l'équipe Runtime d'INRIA Bordeaux, j'ai donc pu profiter la ressource de recherche et les équipements de expérimentation de très haute qualité.

Je remercie chaleureusement les membres de mon jury de thèse. Je remercie tout d'abord mes rapporteurs Jean-François Mehaut et Pierre Sens pour avoir pris le temps d'évaluer mon travail. Je remercie Jean Roman et Patrick Carribault pour avoir accepté mon invitation et participer au jury.

Je remercie tous les membres et les doctorants de département informatique de TELECOM SudParis. Un grand merci à François Trahay, Gaël Thomas et Christian Parrot qui m'ont beaucoup aidé et m'ont beaucoup inspiré pendant la préparation de thèse. Un énorme merci à Brigitte Houassine qui m'a aidé sur tous les démarches administratives. Je remercie Alda Gancarski, Chantal Taconet, Denis Conan, Sophie Chabridon, Olivier Berger, Christian Bac, Amel Mammar et tous les autres que je n'ai pas cités ici. Je remercie tous les membres de l'équipe Runtime de Inria Bordeaux, particulièrement, merci à Denis Barthou, Samuel Thibault, Marie-Christine Counilh et Sylvain Henry pour m'avoir enseigné les reconnaissances sur StarPU runtime.

Je remercie mes anciens collègues Rachid Habel, Alain Muller, Soufiane Baghadadi pour avoir partagé leurs connainssances sur la compilation. Un énorme merci à tous mes co-bureaux qui m'ont supporté pendant ces trois ans. Je également remercie Fabienne Jézéquel et Mounira Bachir qui m'ont encadré mon stage de Master et m'ont proposé à mon encadrant actuel.

Enfin, Je remercie les membres de ma famille pour leur aide et soutien. J'ai commencé mes études en France depuis le 3 septembre 2009. Pendant ces 6 ans, j'ai eu très peu de l'occasion de leur rendre visite, mais vous m'avez toujour supporté et m'avez encouragé. Vous êtes toujour les personnes plus importantes dans ma vie.

Résumé

Résumé en français :

Les travaux de recherche présentés dans cette thèse se positionnent dans le domaine du calcul haute performance ; plus particulièrement dans la démocratisation de l'exploitation efficace des plates-formes de calcul hétérogènes. En effet, les exigences de performance des applications de simulation scientifique mènent à une quête perpétuelle de puissance de calcul. Actuellement, le paysage architectural des plates-forme est tourné vers l'exploitation de co-processeurs, tels les GPU et les Xeon Phi, matériel satellite du processeur principal aux performances surpuissantes sur des cas d'utilisation idoines. Depuis 2007, les GPU (pour *Graphical Processing Unit*) intègrent des milliers de coeurs au design peu sophistiqué capables de traiter efficacement simultanément des milliers de tâches. Plus récemment est apparu le Intel Xeon Phi, co-processeur qualifié de *many-core* car il possède plus de coeurs, plus de threads et des unités d'exécution vectorielles plus larges que le processeur Intel Xeon, son homologue standard. Les coeurs du Intel Xeon Phi sont certes moins rapides si on les considère individuellement mais la performance cumulée est bien supérieure si l'ensemble des ressources est correctement mobilisée à l'exécution d'une application parallèle. Le duo de tête Tianhe-2/Titan du Top500 de ces deux dernières années, classement recensant les 500 machines les plus puissantes, atteste cette tendance : Tianhe-2 est un super-calculateur hétérogène composé de 32.000 processeurs Intel Xeon et de 48.000 co-processeurs de type Xeon Phi, tandis que Titan voit ses 18688 AMD processeurs secondés par 18688 Nvidia Telsa GPU.

Au niveau applicatif, l'exploitation conjointe de ces ressources de calcul aux profils hétérogènes est un réel défi informatique que ce soit en terme de portabilité logicielle, aux vues de la diversité de modèles de programmation de chaque matériel, ou de portabilité de performances avec notamment les coûts de déport de calcul sur de telles ressources. La portabilité logicielle pourrait passer par l'utilisation de standards de programmation tels OpenCL ou OpenACC, qui permettent d'exploiter conjointement l'ensemble des ressources d'une machine, à savoir les processeurs principaux et leurs co-processeurs. Cependant, leur modèle de programmation est statique. C'est à l'utilisateur de décrire quel calcul exécuter sur quelle ressource. L'équilibrage de charge entre les ressources hétérogènes est donc laissé à la charge du programmeur. Ainsi, même si la portabilité logicielle est assurée d'une plate-forme à l'autre, le changement du nombre de ressources ou de leur capacité de calcul impliquent le re-développement de l'application. Il existe des environnements d'exécution qui interfacent les différents co-processeurs et prennent en charge la

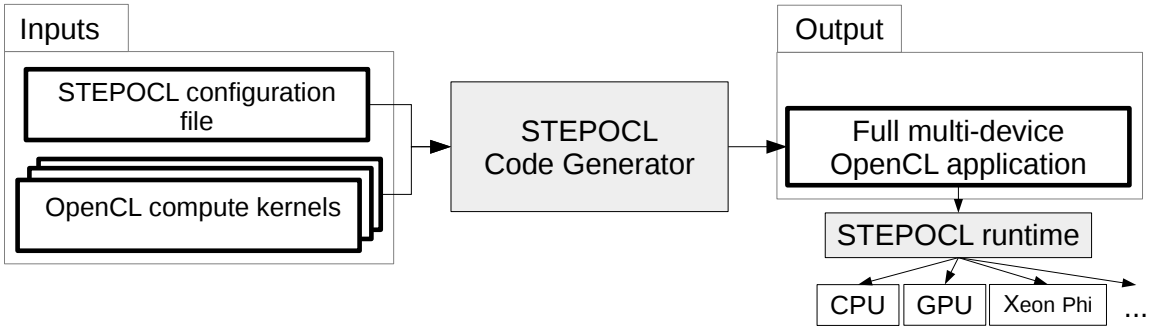


FIGURE 1 – Vue globale de STEPOCL.

dimension équilibrage de charge tels StarPU [6].

Cependant, les outils existants ne reconsidèrent pas la granularité des tâches de calcul définies par le programmeur alors que les données à traiter sont toujours massives et qu'il est de plus en plus fréquent d'avoir plusieurs co-processeurs au sein d'une même machine. Dans ce contexte, il devient intéressant de considérer la distribution d'un calcul sur plusieurs ressources de calcul hétérogènes, d'autant que les calculs adaptés à une exécution sur co-processeur sont généralement massivement parallèles. Plusieurs aspects non-fonctionnels sont à prendre en charge par le programmeur comme déterminer le partitionnement de charge de travail suivant les capacités de calcul des ressources devant participer au traitement, le maintien de la cohérence des données, l'échange de données intermédiaires, etc. Leur mise en ouvre de manière portable et efficace est indéniablement fastidieuse et sujette à erreur pour un programmeur même expérimenté. C'est pourquoi l'objectif de cette thèse est de proposer une solution de programmation parallèle hétérogène qui permet de faciliter le processus de codage et garantir la qualité du code. Nous proposons ici un nouvel outil STEPOCL qui comporte deux volets, comme illustré par la Figure 3.6 : un générateur de code conçu pour générer une application OpenCL complète capable d'exploiter des architectures hétérogènes à partir de noyaux de calcul et de leur description basique écrite grâce à un DSL (*Domain Specific Language*) ; un environnement d'exécution capable de gérer dynamiquement des problèmes comme l'équilibre de charge, la distribution de noyaux de calcul sur plusieurs co-processeurs, la gestion des communications et synchronisation, maintien de la cohérence de données, etc.

Le générateur de code de STEPOCL prend en entrée des noyaux de calcul écrits en OpenCL et un fichier de configuration écrit grâce à un DSL basé sur XML afin de générer une application OpenCL complète faisant appel au support d'exécution de STEPOCL. Le DSL de STEPOCL permet de décrire tous les aspects non-fonctionnels des calculs à réaliser comme la façon dont doivent être subdivisées les données (par exemple, si elles doivent l'être et si oui, suivant quels axes), leur taille, le flot de contrôle attendu entre les différents noyaux de calcul, etc. dans l'application à générer. Le code source généré est un application OpenCL capable d'exploiter plusieurs *device* OpenCL, code que le programmeur est ensuite libre de modifier. Après une phase d'initialisation de l'environnement OpenCL, le code commence par la détection des ressources de calcul effectives auxquelles sont associées un facteur de performance, facteur positionné grâce à un module d'échantillonnage hors ligne. Vient ensuite la phase de déploiement des données

et des calculs sur les ressources sélectionnées. La distribution des calculs est déterminée à partir de la taille des données à traiter, de la forme de partitionnement des calculs donnée dans le fichier de configuration et des capacités de calcul de chacune. Les données nécessaires au calcul de chaque partition sont identifiées grâce à une analyse polyédrique des accès aux données assurée par le compilateur PIPS avant d'être copiées dans leurs mémoires locales respectives. Les données sont par la suite maintenus au maximum en mémoire locale. Seules les données frontières sont communiquées aux ressources qui en ont besoin. Ces données frontières sont également identifiées grâce aux analyses de données produites par PIPS et sont transférées grâce aux mécanismes de copies d'OpenCL via la mémoire globale. Une fois les calculs achevés, les données sont collectées en mémoire globale afin de produire le résultat. Au delà du calcul des régions suivant le nombre effectif de participants au calcul et de l'échange des données au cours de l'exécution, le support d'exécution de STEPOCL s'occupe d'équilibrer dynamiquement la charge de travail des applications itérant sur un même noyau de calcul en monitorant le temps d'exécution de chaque itération sur chaque ressource de calcul.

STEPOCL a donné lieu à deux publications d'intérêt qui ont mis en relief bien des pistes d'amélioration et perspectives à long terme. En premier lieu, il s'agira de pousser la simplification à l'extrême du fichier de configuration en utilisant des outils d'analyse de flot de contrôle à la compilation et des outils de profiling d'exécution permettant d'affiner la granularité des noyaux de calcul. À plus long terme, le contexte d'utilisation de STEPOCL pourra être élargi afin de cibler des objectifs applicatifs différents, comme la réduction d'énergie, avec l'utilisation d'autres stratégies d'ordonnancement et le passage à des architectures différentes.

STEPOCL a été évalué sur trois cas d'application classiques : un stencil 2D 5 point, une multiplication de matrices et un problème à N corps, chacun présentant des noyaux de calcul adaptés à une exécution sur accélérateurs de type GPU ou Xeon Phi car fortement parallèles. En terme de répartition des données, le stencil et la multiplication de matrice sont des cas assez similaires dans le sens où les données vont pouvoir être distribuées sur les ressources ; alors que dans le cas du N-Body, la réplication de la structure de données stockant les particules est nécessaire. Pour chacun de ces cas test, différents points d'évaluation ont été considérés. En premier lieu, est comparé le volume du code généré par STEPOCL à partir d'un noyau de calcul donné et du fichier de configuration associé afin de produire une application OpenCL *multi-device* complète en comparaison de celui de l'application *mono-device* de référence tirée de benchmarks de la littérature. Le fichier de configuration étant massivement plus court et plus simple à écrire que toute la machinerie nécessaire à l'utilisation d'un environnement OpenCL, STEPOCL simplifie le cycle de développement d'une application OpenCL complète en apportant en supplément une dimension multi-device. STEPOCL est ensuite évalué suivant des critères de performance en temps d'exécution. Les évaluations ont été menées sur deux plates-formes matérielles hétérogènes différentes. La première Hannibal allie la force de calcul de trois GPU de type NVidia Quadro FX5800 à un biprocesseur quad-coeur Intel Xeon X5550 ; tandis que sur la seconde, le surpuissant processeur principal, un bi-processeur Intel Xeon E5-2670 comptant 2 fois 10 coeurs de calcul, est équipé de deux accélérateurs de type Intel Xeon Phi offrant chacun 61 coeurs. Chacune de ces ressources est exploitée au travers de l'implémentation OpenCL fournie par Intel à l'exception des GPU qui est adressé grâce au support OpenCL dédié de NVidia. Tout d'abord, l'application produite par STEPOCL

est comparée à sa version de référence dont le noyau de calcul a été extrait et utilisé pour la génération du code. Pour les trois cas tests, les performances du code OpenCL généré par STEPOCL s'exécutant sur un seul et même accélérateur sont comparables à celle de la version originale. Ainsi, STEPOCL permet de produire un code OpenCL complet aux performances satisfaisantes ayant le potentiel de distribuer les calculs sur plusieurs ressources et ce, avec un effort de programmation moindre. Les codes générés sont ensuite exécutés sur les ressources hétérogènes des plates-formes de test. Chaque application est évaluée sur des configurations où seul le processeur principal est activé, puis seul un co-processeur, puis deux pour enfin arriver à l'exploitation totale des ressources hétérogènes de la machine. L'équilibrage de charge de calcul réalisé à partir de l'échantillonnage hors-ligne de STEPOCL permet d'exploiter de manière conjointe toutes les ressources dans les trois cas applicatifs. De plus, dans le cas du stencil et de la multiplication de matrices, le fait que les données puissent être distribuées sur les différentes ressources permet de traiter des jeux de données plus larges en parallèle. Avec l'application originale, seule une exécution sur le processeur hôte de la plate-forme Hannibal permettait de mener à bien le calcul car la taille de la mémoire des GPU est trop limitée.

Mots-clés : Calcul Haute Performance, Parallélisme, Architectures hétérogènes, OpenCL, génération de code, équilibrage de charge

Résumé en Anglais:

Heterogeneous architectures have been widely used in the domain of high performance computing. However developing applications on heterogeneous architectures is time consuming and error-prone because going from a single accelerator to multiple ones indeed requires to deal with potentially non-uniform domain decomposition, inter-accelerator data movements, and dynamic load balancing.

The aim of this thesis is to propose a solution of parallel programming for novice developers, to ease the complex coding process and guarantee the quality of code. We lighted and analysed the shortcomings of existing solutions and proposed a new programming tool called STEPOCL along with a new domain specific language designed to simplify the development of an application for heterogeneous architectures. We evaluated both the performance and the usefulness of STEPOCL. The result show that: (i) the performance of an application written with STEPOCL scales linearly with the number of accelerators, (ii) the performance of an application written using STEPOCL competes with an handwritten version, (iii) larger workloads run on multiple devices that do not fit in the memory of a single device, (iv) thanks to STEPOCL, the number of lines of code required to write an application for multiple accelerators is roughly divided by ten.

Keywords: High-Performance Computing, Parallelism, Heterogeneous Architectures, OpenCL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Que peu de temps suffit pour changer toutes les choses.

—*Victor Hugo*

Contents

Since the first general-purpose electronic computer ENIAC was created in 1946, scientists have never stopped the pace of creating new computers with higher performance and computational power. Scientists had kept enhancing the performance by increasing CPU frequency and complexity of CPU architecture until they meet the bottleneck of heat dissipation. Meanwhile, integrated circuit transistor technology has almost reached its physical limit. The number of transistors contained in processor chip can be used as a rough estimate of its complexity and performance. *Moore's law* which is an empirical observation states that the number of transistors of a typical processor chip doubles every 18 to 24 months. However today, there is not much space for increasing the density of transistors in a single CPU chip, because extreme miniaturization of electronic gates makes the effects of phenomena like electromigration and subthreshold leakage become much more significant [24]. These factors make scientist to investigate new solutions: parallelism.

Instead of using single high frequency unit, the trend of computer architecture has turned to use more but relatively slow processing units (multi-core processors). In the early 2009s, most desktop CPUs have become multi-cores. Since then, more and more types of multi-core processor have been used in general purpose computing and heterogeneous architectures which are usually composed of a host multi-core processor (CPU) and some auxiliary specially designed processors (called accelerators, such as GPU and Intel Xeon Phi) have become very important in the domain of high performance computing. In Top500 [5] fastest supercomputer lists of 2015, both Tianhe-2 (ranked as No. 1) and Titan (ranked as No. 2) used different type of processors: Tianhe-2 consists of 32,000 Intel Xeon CPUs and 48,000 Intel Xeon Phi coprocessors; Titan consists of 18,688 AMD Opteron CPUs and 18,688 Nvidia Tesla GPUs.

Compared to homogeneous architectures, heterogeneous architectures have greater advantage on computing performance. These systems gain performance not just by adding cores, but also by incorporating specialized processing capabilities to handle particular tasks. Heterogeneous architectures utilize multiple processor types to benefit the best of each devices. For example, GPU processing, apart from its well-known 3D graphics rendering capabilities, can also perform mathematics computations on very large data sets, while CPUs can run the operating system and perform traditional serial tasks, such as data transfer management. Moreover, GPU and Xeon Phi have vector processing capabilities that enable them to perform parallel operations on very large sets of data at much lower power consumption relative to the serial processing of similar data sets on CPUs. Followed with the rise of heterogeneous architectures, applications such as augmented reality, rich media composition, numerical simulations makes heterogeneous architectures usage more productive.

However the development of such applications is a challenge due to the usage of various hardware and many APIs (application program interface) together with a goal for power-performance efficiency. Designing applications for multi-accelerator heterogeneous systems requires that developers have rich experience of parallel programming and some strong knowledge of architectures. Firstly, developers need to choose the most appropriate set of computing devices for a given application. Then they have to write compute kernels optimized for the selected accelerators. In order to achieve the best possible performance, the developer has to identify an efficient partitioning of the workload among the accelerators. This efficient partitioning is not only related to the theoretical performance of the accelerator, but also to the actual application and its workload. Then, the developer has to write code to coordinate the execution of multiple kernels running on different devices, to partition the workload among them and to perform data movements and synchronization between them. Implementing these features is time-consuming and error-prone. Moreover, the developer often implements these features for a specific hardware and has to drastically modify his code for a different target.

As we will present in this documents, existing programming tools for heterogeneous architectures typically define APIs to deploy and execute *compute kernels* on the processing units of accelerators, however, none of these programming tools are tailored for multi-accelerator application development. The challenges mentioned above are still left for developers to resolve.

1.1 Objective of thesis

The main objective of this thesis is to study heterogeneous computing at programming level and to propose a unified system that could automatically generate code for heterogeneous multi-device architectures. We want to relief the developers from tedious and time consuming coding process, and make them more focus on solving the problem itself.

More specifically, this thesis will focus on studying how to solve following problems:

- How to split and distribute a parallel task to multiple devices;
- How to automatically balance the workload of whole system and manage the communications between devices;

- How to automatically generate the whole application without too much coding effort.

1.2 Outline and contribution

Chapter 2 introduces the state of the art of heterogeneous computing including the computing devices equipped on current heterogeneous architectures and programming tools for developing applications.

In Chapter 3, we first introduced the background of OpenCL which is the basis of our main work. Then we presented our contribution: a new programming tool called STEPOCL. STEPOCL separates the functional aspects of the code, *i.e.*, the compute kernels, from the non-functional ones. The non-functional aspects of the code are described in a domain specific language. It mainly consists in describing the data layout and the kernel of the application. Using a domain specific language improves the productivity of the developer by decreasing the number of lines of codes required to implement a full multi-device application. Moreover, the code written in the domain specific language is not limited to a specific hardware setting, which increases the portability of the code. Finally, it also has the advantage of avoiding many errors caused by the use of a low-level language. Based on the compute kernels and on the description of the non-functional aspects, STEPOCL automatically generates a full OpenCL application, but also an offline profiler for this application. The profiler computes an efficient workload partitioning for a specific set of accelerators and a specific workload. The generated application handles the initialization of the OpenCL environment, which includes the discovery of the accelerators, the mapping of the data to the accelerators according to the offline profiling results, and the launch of the kernels over the accelerators. During execution, the generated application also automatically exchanges the data between the accelerators to maintain the data consistency thanks to a polyhedral data analysis, and, at the end of the execution, the generated application retrieves the results from the accelerators.

Chapter 4 introduces the implementation of main modules of STEPOCL including analysis of region, offline profiling, runtime system and code generation. These modules ensure that the generated applications can profit the full potential of any multiple devices heterogeneous architectures.

Chapter 5 presents the evaluation of generated applications. We evaluate STEPOCL with three application kernels (a 5-point 2D-stencil, a matrix multiplication and an N-body application) on two multi-device heterogeneous machines that combine CPUs with different accelerators: CPU+GPUs and CPU+Xeon Phis. Our main results show that:

- When running on multiple heterogeneous devices [1], the performance of the code generated by STEPOCL scales linearly with the number of devices.
- As compared to the same applications written directly in OpenCL and provided with the OpenCL version of AMD, the applications written with STEPOCL have similar performance.
- Thanks to STEPOCL, we are able to run large workloads on multiple devices that do not fit in the memory of a single device.

[1] A device is either a CPU or an accelerator.

- As compared to the generated code, STEPOCL is able to divide by ten the number of lines of code of the application. Also as compared to the native OpenCL applications, STEPOCL is able to divide by five their number of lines of code. Furthermore, while the applications shipped with the OpenCL version of AMD only runs on a single device, the STEPOCL applications are able to run on multiple heterogeneous devices.

At last, chapter 6 concludes the whole works of this thesis and presents future work.

Chapter 2

The rise of heterogeneous computing

Essayez de ne pas devenir un homme de succès, mais plutôt essayez de devenir un homme de valeur.

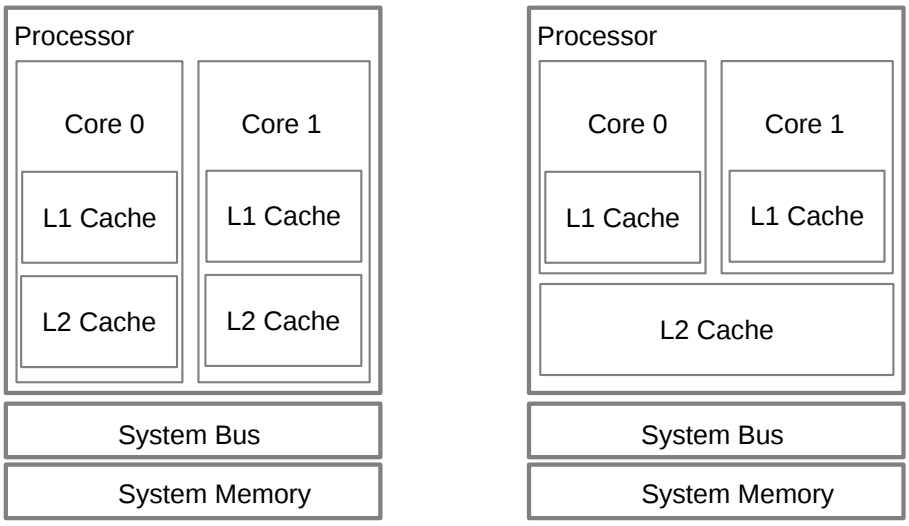
—*Albert Einstein*

Contents

This chapter introduces heterogeneous computing from two aspects: hardware architecture and programming on heterogeneous architectures.

2.1 Heterogeneous platforms

A computer is usually equipped with two types of processor: CPU (central processing units) and GPU (graphics processor unit). Before the 1990s, CPUs still play important role in massive computing while GPUs are merely displaying the monitor. In order to improve the performance of computers, engineers focused on improving the performance of CPU mainly by increasing in the frequency and complicating the architecture (mainly by using more transistors on a CPU die). However, if the frequency gets too high, the



(a) Multi-core processor seperate L2 (b) multi-core processor shared L2

Figure 2.1 – A basic diagram of a dual-core processor

chip will melt by the extreme heat. That's the reason why most current CPUs runs about 2 GHz to 3 GHz, but not more than 4GHz. Meanwhile, due to the physical limit, increasing the density of transistor on a single chip also meet the bottleneck.

2.1.1 Multi-core processor

In order to keep improving the performance over the last 10 years, engineers found other ways – using more CPU cores on the same CPU chip. Figure 2.1 presents two typical multi-core processor architectures. The advantage of multi-core processors is that multiply tasks (such as running a software or scientific computation) can be dispatched to different CPU core, each core can work at relatively low frequency but we can still get a faster experience. The dual core CPUs and quad core CPUs on the market today are good enough for regular tasks in daily life. However, for scientists and engineers, CPUs are still not powerful enough. Meanwhile, GPUs, due to their special architecture and extraordinary data parallel processing ability, have become very important in the domain of high performance computing.

2.1.2 GPU computing

The original GPUs were designed to rapidly render images in a frame buffer intended for output to a display. In the last 20 years, the improvement of GPUs have been primarily driven by the demand for realtime, high-definition graphic, the evolution of GPU hardware architecture has gone from a specific single core, function hardware pipeline implementation made only for graphics, to a set of parallel and programmable cores for more general purpose computations [22]. Modern GPU has become a highly parallel,

multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 2.2 and Figure 2.3.

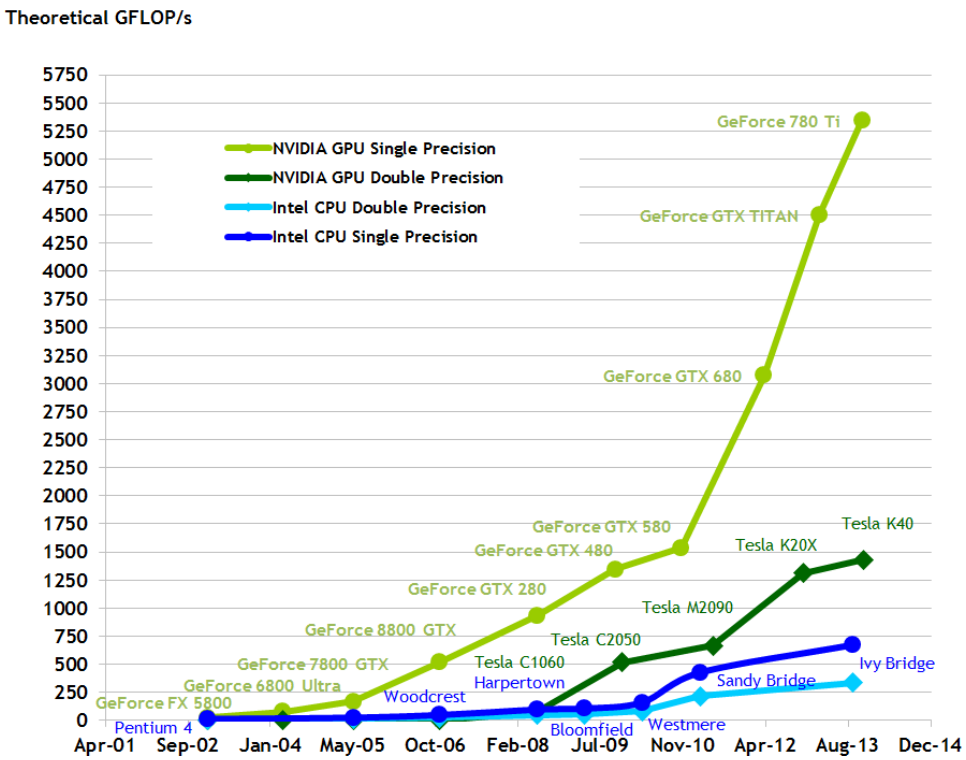


Figure 2.2 – Floating-Point Operations per Second for the CPU and GPU.

The reason behind the difference of floating-point capability between the CPU and the GPU is that the architecture of modern GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as illustrated by Figure 2.4.

More specifically, the GPU is well designed for dealing with the problems that can be expressed as data-parallel computations (the same program is executed on many data elements in parallel). Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Many applications that process large data sets can use a data-parallel programming model to speed up the computations, from image rendering or signal processing to physics simulation or computational biology. The success of GPU computing opened a new page of heterogeneous computing, and new accelerators (such as Intel Xeon Phi [26]) begin to emerge.

2.1.3 Intel Xeon Phi

Intel Xeon Phi Coprocessor is the brand name for all Intel Many Integrated Core Architecture (Intel MIC Architecture) based products. The Intel Xeon Phi coprocessors are

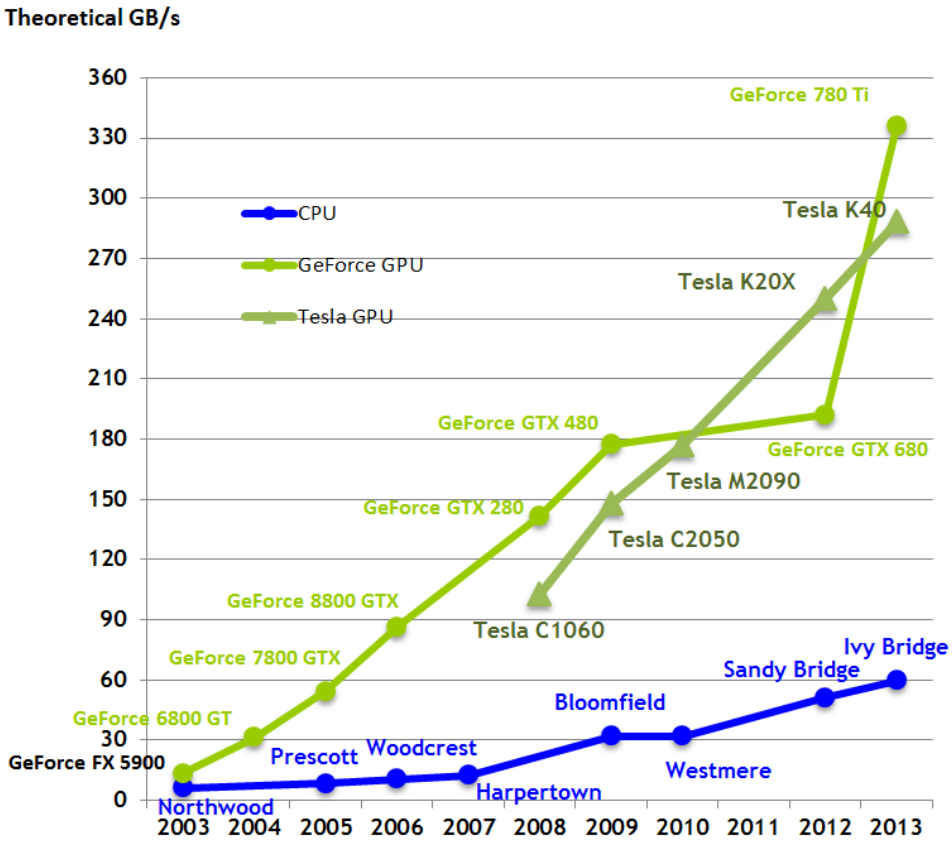


Figure 2.3 – Memory bandwidth for the CPU and GPU.

designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and fully exploit available processor vector capabilities or memory bandwidth. Compared to traditional Intel Xeon processor, the Intel Xeon Phi coprocessors provide extra power-efficient scaling, vector support, and local memory bandwidth. Each Intel Xeon Phi coprocessor equips more than 50 cores (it varies between generations) interconnected by a high-speed bidirectional ring as presented in Figure 2.5. Each core has four hardware threads and clocked at 1 GHz or more. The many core architecture makes Intel Xeon Phi coprocessors competitive with GPUs on the performance of parallel computing. Moreover, Intel Xeon Phi coprocessor offers more programmability. Applications that show positive results with GPUs should always benefit from Intel Xeon Phi coprocessors because the same fundamentals of vectorization or bandwidth must be present. The opposite is not true, the flexibility of an Intel Xeon Phi includes support for applications that cannot run on GPUs [18].

2.1.4 AMD APU

Current CPUs and GPUs have been designed as separate processing elements, each has a separate memory space. In order to launch an execution on GPU, the required data located on CPU memory needs to explicitly be copied to GPU memory and then copied

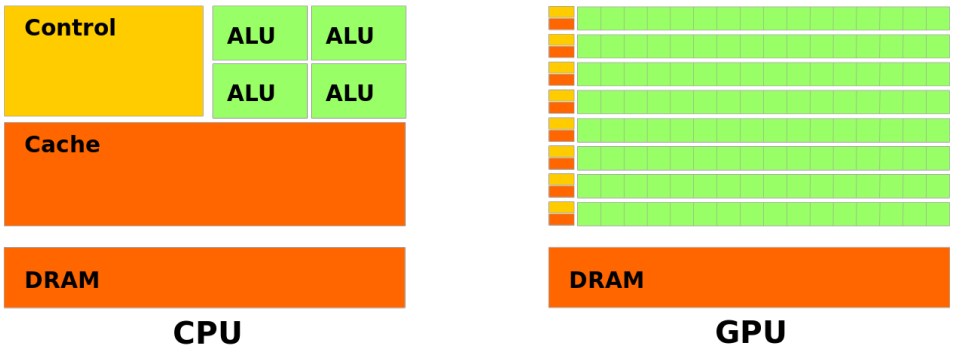


Figure 2.4 – The GPU devotes more transistors to data processing.

back again. AMD Accelerated Processing Unit (APU) used **Heterogeneous System Architecture (HSA)** that integrates CPU and GPU on the same bus with unified shared main memory. As a result, APU performs zero data movement between CPU, GPU and other accelerators. The design of unified memory reduces communication latency between CPU and other accelerators. Moreover, unified memory is more practical on programming level.

2.2 Exploitation of heterogeneous architectures

Since heterogeneous architectures become more and more popular, the computing community is building tools and libraries to ease the use of these heterogeneous systems.

2.2.1 Computing on CPUs

The parallelism of a computer program on multi-core CPUs can be achieved through using threads model. In threads model, a parallel main program can have multiple concurrent execution paths, and each path is executed by a thread. In Figure 2.6, program **main.out** initializes two arrays, and then creates four tasks (threads), each tasks will be scheduled and run by operating system concurrently. Each thread has local data and shares a global memory. The synchronization mechanism are usually applied to avoid the conflict of operation on same global memory address, such as avoiding that several threads update the same global address at same time.

Pthreads

Pthreads is a POSIX (Portable Operating System Interface) standard [4] for threads. It provides an API (Application Programming Interface) for creating and manipulating threads. **Pthreads** is implemented in a pthread.h header and a thread library. Although **Pthreads** only supports C language, it is available on many mainstream operating systems such as FreeDSB, Linux, Mac OS x and Windows.

The example in Listing 2.1 creates 4 threads with *pthread_create* function, and each thread prints its own *id*. Listing 2.2 shows the execution result in Listing 2.2.

Listing 2.1 – Pthreads application in C

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

void *print_thread_id(void *argument)
{
    long thread_id;
    thread_id = (long ) argument;
    printf("I am thread %ld!\n", thread_id);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t thread[NUM_THREADS];
    int err;
    long t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %ld!\n", t);
        err = pthread_create(&thread[t],NULL,print_thread_id,(void *)t);
        if(err){
            printf("ERROR!\n");
            exit(-1);
        }
    }
    pthread_exit(NULL)
}
```

Listing 2.2 – Execution result

```
Creating thread 0!
Creating thread 1!
I am thread 0!
Creating thread 2!
Creating thread 3!
I am thread 1!
I am thread 2!
I am thread 3!
```

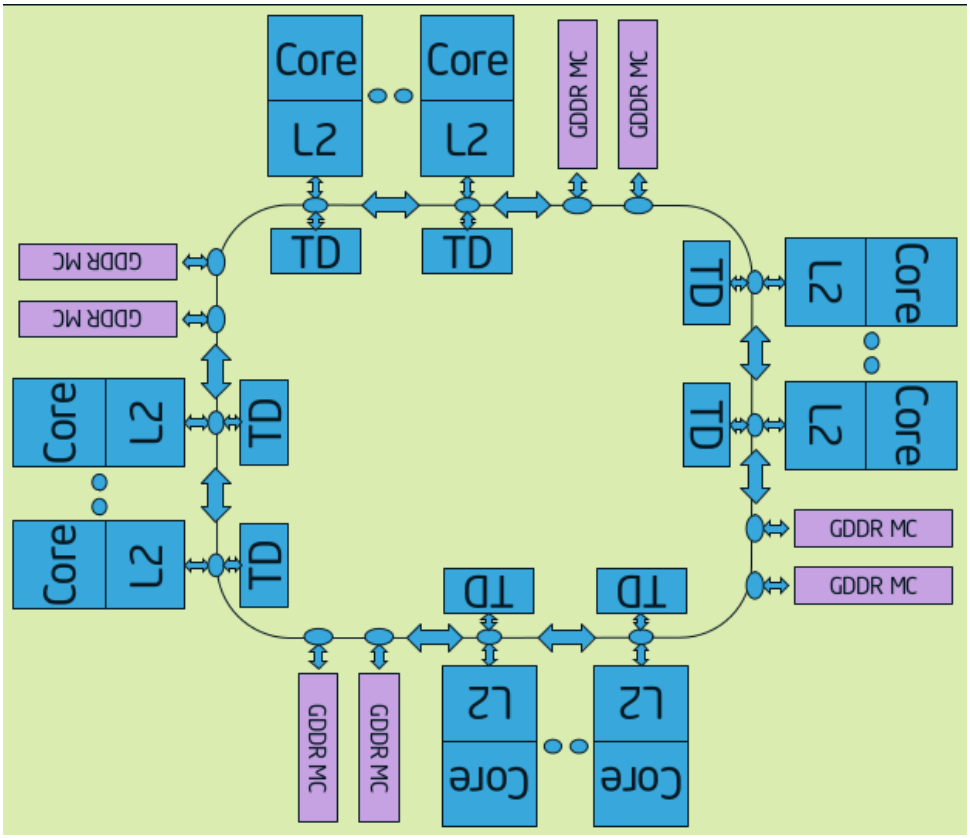


Figure 2.5 – Intel Xeon Phi microarchitecture.

OpenMP

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. As an industrial standard, OpenMP can run on most platforms including Linux, Mac OS X and Windows platforms. OpenMP program is composed of a set of compiler directives, runtime library routines and environment variables. OpenMP uses the fork-join model of parallel execution. At the beginning, OpenMP program starts as a single thread: the master thread. The master thread executes sequentially until it reaches the parallel region construct. The master thread then creates a group of parallel threads to execute the statements that are enclosed in parallel region construct. Once the group of threads finish the execution of the statements in parallel region construct, they will be synchronized and deleted, leaving only a master thread.

OpenMP directives can be inserted directly into serial code and achieve a considerable improvement of performance. Listing 2.3 presents a program that initiates a table with multiple threads. The number of threads used can be defined as environment variables. Listing 2.4 presents an example of setting OpenMP environment. In this case, eight threads will be used to initialize in parallel *tableA* of Listing2.3.

Listing 2.4 – Setting OpenMP environment variables

```
export OMP_NUM_THREADS=8
```

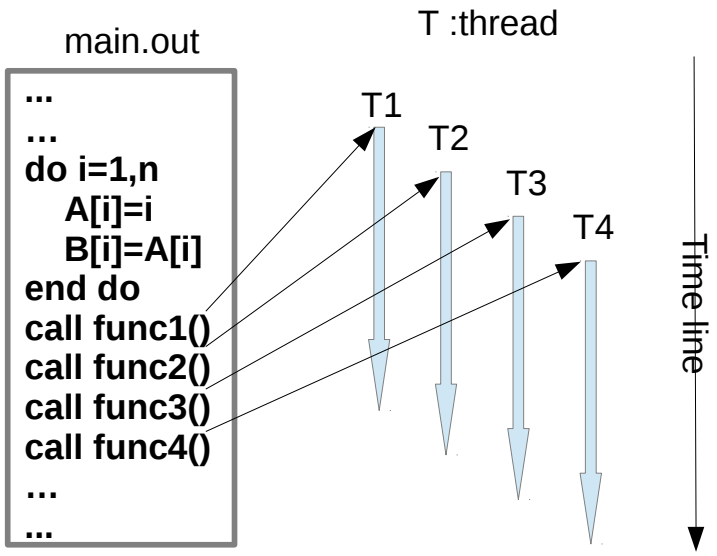


Figure 2.6 – Theads model

OpenMP is well suited for shared memory multi-core architectures. However, if programmers want to use more computing resources such as multiple nodes on large-scale clusters, they have to use other APIs, such as MPI [11] (Message Passing Interface), to manage computation and communication on distributed memory environment.

STEP and dSTEP

STEP [23] is an OpenMP directives based programming tools that transforms a program into an MPI source program and produces MPI code close to hand-written MPI code. It is developed by HP2 (Haute Performance et Parallélisme) team of TELECOM SudParis. The programmer adds OpenMP directives and then STEP generates a MPI program automatically. STEP extends the usage of OpenMP (which is restricted to

Listing 2.3 – Initializing a table in parallel

```
int main(int argc, char* argv[]){
    const int size = 100;
    int tableA[size];

    #pragma omp parallel for
    for(int n=0; n<size; n++)
        tableA[size] = n*n

    ... ...
}
```

shared-memory architectures) to distributed-memory architectures such as clusters of many-cores. Thanks to the region analysis of compiler PIPS [16], STEP can efficiently generate the communications between devices. Habel et al. improved the data partitioning module of STEP, and named the new tool as dSTEP [13]. dSTEP provides a large set of distribution types for data partitioning. These distribution types are unified in a ”dstep distribute” directive. It also provide a ”dstep gridify” directive to express the computation distribution and the schedule constraints of loop iterations.

At the beginning of this thesis, the main limitation of STEP and dSTEP is that they do not fully support the computation on the accelerators such as GPUs and Intel Xeon Phi (dSTEP supports CUDA). Besides, the data partitioning of dSTEP is homogeneous: every sub-data has same size. Considering the varieties of hardware performance of computing devices, a homogeneous data partitioning may lead to poor performance.

2.2.2 Computing on accelerators

Some programming platforms provide accessibility to multiple type accelerators such as GPU and co-processors. Different from multi-core CPU architecture, the memory on the accelerator may be completely separated from main memory (which is shared by multi-core CPU). Most of programming platforms for diverse computing resources use an *offload* model which includes a single **host** and one or more **accelerators** which can be any kind of processor (CPU, GPU and Xeon Phi etc.). Figure 2.7 presents a processing flow of an *offload* model. First of all, developers allocate data region on main memory (host memory). Then data region are transferred from main memory to remote memory on accelerators. In the third step, the host instructs the process to accelerators, accelerators execute computing functions submitted from host. At last, the results are copied to main memory. Due to the fact that accelerators do not share a common memory, programmers have to manually maintain the consistency of data in each remote memory.

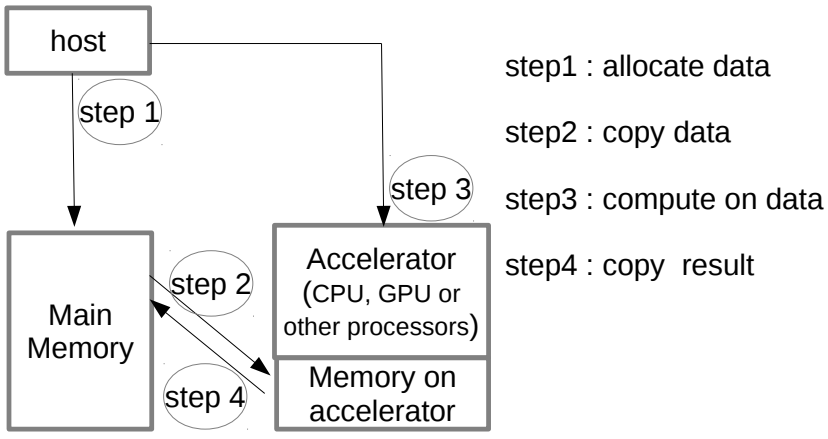


Figure 2.7 – Host-accelerator model

Listing 2.5 – The syntax of OpenACC directives

```
#define SIZE 1000
float a[SIZE];
float b[SIZE];

int main()
{
  int i;
  // Initialize arrays.
  for (i = 0; i < SIZE; ++i) {
    a[i] = (float)random()%1000;
    b[i] = 0.0f;
  }

// 1D stencil computing
  #pragma acc kernels copyin(a) copy(b)
  for (i = 1; i < SIZE-1; ++i) {
    b[i]=(a[i-1]+a[i]+a[i+1])/3
  }
  return 0;
}
```

CUDA

CUDA (Computer Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA [1]. It allows software developers to benefit the computing performance of CUDA-enable GPU for general purpose processing.

The CUDA platform is composed of CUDA libraries and compiler directives. CUDA supports several industry-standard programming language including C, C++ and Fortran. Unlike OpenMP which achieves parallelism by inserting directives on serial code, CUDA users need to define specific functions (called kernel) for GPU computing. A kernel is executed by a number threads, each having a unique *threadID*. CUDA organizes threads into a hierarchy of a grid of thread blocks. A grid contains a set of thread blocks with unique *bockID*, and each blocks contains same amount of threads. The threads in a block can access to a shared memory space private to the block and can be synchronized through synchronization function.

CUDA supports most of operating systems such as Microsoft Windows, Linux and Mac OS. However, CUDA only works with Nvidia GPUs.

OpenACC

OpenACC (Open Accelerators) [29] is a programming standard for programming on heterogeneous CPU/GPU systems in C/C++ and FORTRAN. OpenACC standard is developed by Nvidia, Cray, CAPS and PGI. Similar to OpenMP, OpenACC provides a set of compiler directives to specify parallel regions, control flow instructions on accelerators and manages data movements on remote memory of accelerators.

Listing 2.5 presents an example of an OpenACC program that performs a 1D 3-points stencil computation with *offload* mode. The OpenACC directives in Listing 2.5 tells the

Listing 2.6 – OpenACC in OpenMP threads

```
... ...
numgpus = acc_get_num_devices( acc_device_nvidia );
#pragma omp parallel num_threads(numgpus)
{
    gpunum = omp_get_thread_num();
    acc_set_device_num( gpunum, acc_device_nvidia );

    #pragma acc data copy( x[0:n] )
    {
      // thread tnum copies x to GPU gpunum
      ... ...
      // launch computation
      ... ...
    }
}
... ...
```

compiler following information:

- *#pragma acc*: This is an OpenACC directive.
- *kernels*: This is a parallel regions (also called kernel region), which contains work-sharing loops.
- *copyin(a)*: Array a needs to copied from the host memory to the device memory.
- *copy(b)*: Array b needs to copied from the host memory to the device memory, and the assigned value of array b on device memory needs to be copied back to the host.

As we can see from Listing 2.5, OpenACC eases the way we program on heterogeneous architectures. However, OpenACC is not suited to multiply device programming. In order to concurrently use multiple GPUs on a machine, programmers have to mix OpenACC with other API (such as OpenMP or MPI [11]). Listing 2.6 presents an example of OpenACC program using multiple Nvidia GPU. By using OpenMP directives, each OpenMP thread uses a different GPU. Then, programmers have to manage workload balancing, data movement and communications with hybrid directives (OpenMP and OpenACC directives) which is quite confusing and error-prone.

OpenCL

OpenCL is designed for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, field-programmable gate arrays (FPGAs) and other processors. As an open standard maintained by Khronos Group, OpenCL is supported by most of hardware manufacturers such as Apple, IBM, Intel, Nvidia, Qualcomm, Samsung, etc. OpenCL is also supported by most computer systems, such as Windows, Linux and Unix.

OpenCL also defines computing functions as *kernels* for the execution on *compute device*. A *work-item* in OpenCL plays the same role as a thread in CUDA. Work-items

are organised into work-groups. OpenCL achieves parallelism by simultaneously executing kernel by each work-item. A main purpose of OpenCL is to unify the programming model of heterogeneous platforms. OpenCL views a processor of any type (CPU, GPU, FGPAs ...) on a machine as a *compute device*. All devices are logically defined a common abstract memory hierarchy. Thus an OpenCL program is portable across different platforms, although the performance has not necessarily the same portability. Our main works in this thesis are based on the OpenCL framework and our primary objective is to improve the portability of performance of OpenCL applications. More details about OpenCL is presented in Section 3.1.

2.2.3 Computing on heterogeneous architectures

Although some programming languages can natively support using multiple computing resources, many studies further provide facility to exploit full power of heterogeneous architecture.

Bolt C++ Template Library

Bolt [2] is an OpenCL–based C++ template library optimized for GPUs. Compared to Standard OpenCL API, Bolt provides a more simplified STL-like interface, it can directly interact with host memory structures such as *std::vector* or host arrays (e.g. float∗). Bolt also provides common optimized routines like sort(), reduce(). The library itself can select automatically where to execute such a routine (GPU or CPU). No OpenCL API calls are required since Bolt library handles all initialization of the OpenCL environment and the communication between devices. Listing 2.7 presents how to use Bolt to sort a random array. However, the usage of Bolt is limited to built-in routines and it is only available on AMD devices.

Listing 2.7 – Sort a random array with Bolt

```
#include <bolt/cl/sort.h>
#include <vector>
#include <algorithm>

int main()
{
    // generate a random array on host
    std::vector<int> array(1024);
    std::generate(array.begin(), array.end(), rand);
    // sort, run on best device in the platform
    bolt::cl::sort(array.begin(), array.end());
    return 0;
}
```

Boost.Compute

Boost.Compute [3] is a GPU/parallel-computing library for C++ based on OpenCL. It provides an STL-like C++ interface which is very similar to Bolt. It contains common algorithms (e.g. transform(), sort()) along with common containers (e.g. vector<T>,

flat_set<T>). In Boost.Compute, the interaction between host and computing devices is managed by an object of the *command_queue* class which corresponds *clCommandQueue* in standard OpenCL. Listing 2.8 presents how to use Boost.Compute to sort a random array. Boost.Compute as well as Bolt is very well suited for implementing data initializations or launching tasks with existed built-in routines. However, expressing some tasks which require more complicated data structures cannot be supported. The studies of Peter et al. [25] also prove that Boost.Compute and Bolt provide high compute performance for simple tasks, but very low performance for complex tasks.

Listing 2.8 – Sort a random array with Boost.Compute

```
#include <vector>
#include <algorithm>
#include <boost/compute.hpp>

namespace compute = boost::compute;

int main()
{
    // get the default compute device
    compute::device gpu = compute::system::default_device();
    // create a compute context and command queue
    compute::context ctx(gpu);
    compute::command_queue queue(ctx, gpu);
    // generate random numbers on the host
    std::vector<float> host_vector(1024);
    std::generate(host_vector.begin(), host_vector.end(), rand);
    // create vector on the device
    compute::vector<float> device_vector(1024, ctx);
    // copy data to the device
    compute::copy(host_vector.begin(), host_vector.end(),
        device_vector.begin(), queue);
    // sort data on the device
    compute::sort(device_vector.begin(),
        device_vector.end(), queue);
    // copy data back to the host
    compute::copy(device_vector.begin(), device_vector.end(),
        host_vector.begin(), queue);
    return 0;
}
```

BOAST

BOAST [10] is an automatic source-to-source transformation tool which optimizes loop structures in order to find the best performance configuration for a given type of computing resource. According to the configurations which are defined by users, BOAST can generate code in C or Fortran. The main interest for our work is the idea of architecture-specific optimization used in BOAST. Due to the difference of hardware features on heterogeneous architectures, the generic methods of code transformation may lead to a poor performance. In order to optimize the performance for a specific architecture, we have to tune the code accordingly. The techniques of code transformation could also be

used for the optimization of OpenCL kernel in our future works.

StarPU

StarPU [6] is a software that helps programmers to exploit the computing power of heterogeneous multi-device architectures. StarPU is a C library. The core component of StarPU is a run-time support library which is used for scheduling the tasks on heterogeneous multi-core systems. An execution of StarPU is defined by two data structures: **codelet** and **task**. A **codelet** describes a computational kernel and indicates runnable architectures such CPUs or GPUs. A **task** describes which data can be accessed by **codelet**. Once a **task** is submitted, StarPU will automatically assess accessibility of computing resources and will schedule the task to ensure load balancing over heterogeneous systems.

StarPU has implemented a rich set of scheduling policies such as **eager**, **random** and **heft** (heterogeneous earliest finish time), etc. These features provide great facility for the implementation of task-parallel applications. However, StarPU is not perfect at partitioning data-parallel tasks. StarPU provides an interface which can partition a data region homogeneously, however a perfect partition should consider the diversity of performance of available computing devices, then partition the data region in appropriate proportion. Moreover, programmers still have to manually manage communication and consistency of data which is time-consuming and error-prone.

StarPU provides an OpenCL extension called SOCL [14]. Unlike a standard OpenCL application, SOCL is not dedicated to a specific kind of device nor to a specific hardware vendor, it virtualizes all existing available devices (CPUs and GPUs) as one unique OpenCL device on a unified OpenCL Platform. Thus, programmers no longer need to create extra OpenCL context and OpenCL command queues for each CPU or GPU, all computing kernel just go to only one command queue which is responsible for the execution on virtual machine. Once computing kernels are submitted, SOCL transparently distributes computing kernels over physical devices at runtime.

As an extension of StarPU, SOCL provides powerful automatic scheduling abilities on heterogeneous architectures and simplified the programming interface of OpenCL. But the key issues are still unsolved. SOCL is still not able to handle workload partition of data-parallel tasks and manage the data consistency between devices.

libWater

libWater [12] is a C/C++ library-based extension of the OpenCL programming model that aims to alleviate programming process on large-scale clusters. libWater simplified the programming interface of OpenCL and introduces a new device query language (DQL) for accessing distributed devices on different compute nodes. Listing 2.9 presents the basic syntax of DQL.

Listing 2.9 – DQL of libWater

```
SELECT          [ALL | TOP k | POS i]
    FROM NODE   [n [, ...]]
    WHERE       [restrictions attribute values]
    ORDER BY    [attribute [, ...]]
```

libWater eliminates and replaces some redundant OpenCL instruction (such as the initialization of OpenCL platforms and contexts) by its own API which allows programmers to more focus on the development of computing kernels. Meanwhile, the distributed runtime system of libWater dispatches the OpenCL commands to the addressed devices and transparently moves data across the cluster nodes. libWater also enhances the event system of OpenCL by enabling inter-context and inter-node device synchronization and improves the efficiency of runtime system by analysing the collected event information. Although libWater provides a convenient interface to access the distributed devices and dispatch the work task on large-scale clusters, it has not yet provided any solution for workload partition which is also a big problem for programmers.

Kim's framework for multiple GPUs

Kim et al.[20] propose an OpenCL framework to manage multiple GPUs within a node. This OpenCL framework combines multiple GPUs as a virtual single GPU. Programmers only need to provide an application for single GPU, the framework takes in charge the deployment of the code on the multiple GPUs transparently by partitioning the workload equally among the different devices. In order to partition the workload precisely, the runtime system applies a run-time memory access range analysis to the kernel by performing a sampling run and identifies an optimal workload distribution for the kernel. The sampling code is generated from the OpenCL kernel code by using a OpenCL-C-to-C translator, it only accesses to memory and performs no computation. Meanwhile, the runtime maintains virtual device memory that is allocated in the main memory and keeps it consistent to the memories of the multiple GPU devices. They also use an OpenCL-C-to-CUDA-C translator to generates the CUDA kernel code for the distributed OpenCL kernel.

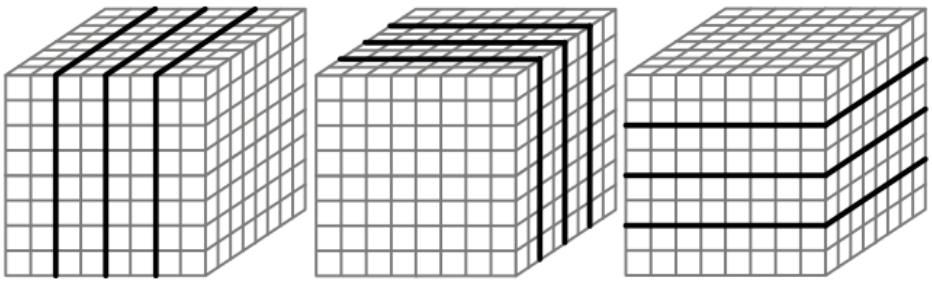


Figure 2.8 – Data Partitioning in homogeneous way

However Kim et al. only focuse on homogeneous multiple device systems and data independent tasks. Their OpenCL framework can not work with heterogeneous devices. Due to the diversity of performance between devices, an equal data partitioning (such as Figure 2.8) can not efficiently exploit full computing power of heterogeneous multi-device systems. Moreover the overhead of sampling run and the overhead of translation from OpenCL-C kernel to CUDA are not negligible.

SKMD

Lee Janghaeng et al. propose a tool, called SKMD, that generates multi-device source code from a single-device kernel [17]. This system transparently orchestrates collaborative execution of a single data-parallel kernel across multiple asymmetric CPUs and GPUs. The programmer is responsible for developing a single data-parallel kernel in OpenCL, while the system automatically partitions the workload across an arbitrary set of devices, generates kernels to execute the partial workloads, dynamically adjusts the workload across devices and merges the partial outputs together.

SKMD implements workload partition through assigning partial work-group on original kernel. As showed in figure 2.9, SKMD adds two parameters *WG_from* and *WG_to* to represent a target range of work-group index to be computed on a device. In other world, each computing device only runs $(WG\_from - WG\_to + 1)$ work-groups of the kernel. At end of the execution, SKMD also utilises these two parameters to retrieve the computing result from discrete device memory.

```
1  __kernel void add_CPU(
2      __global float *input,
3      __global float *output,
4     )
5  {
6
7     int tid = get_global_id(1)
8           * get_global_size(0)
9           + get_global_id(0);
10
11    output[tid] = input[tid]+output[tid];
12 }
```

```
1  __kernel void add_CPU(
2      __global float *input,
3      __global float *output,
4      int WG_from, int WG_to)
5  {
6      int idx = get_group_id(0);
7      int idy = get_group_id(1);
8      int size_x = get_num_groups(0);
9      int tag_id = idx + idy * size_x;
10     // check whether to execute
11     if (tag_id < WG_from || tag_id > WG_to)
12        return;
13
14     int tid = get_global_id(1)
15          * get_global_size(0)
16          + get_global_id(0);
17
18     output[tid] = input[tid]+output[tid];
19 }
```

Figure 2.9 – Comparison of original kernel and partitioned kernel

To do so, SKMD duplicates all data on every device. Thus each device needs to allocate all the data, even if the device will not use it. This unnecessary memory redundancy for the whole system limits the problem size.

InsiemeTP

Klaus Kofler et al. [21] propose an approach to automatically optimize task partitioning for different problem sizes and different heterogeneous architectures. They use the Insieme[19] source-to-source compiler to translate a single device OpenCL program into a multi-device OpenCL program.

The architecture of their framework consists of two main phases: *training* and *deployment*. At the training phase, a set of applications is executed with different partitioning

strategies. The statistics (including static program features, runtime feature and the time of execution) of each execution is used for building a task partitioning prediction model. The prediction model is based on Artificial Neural Networks approach. At the deployment phase, the Insieme compiler analyse the static feature of input code and translate it into a multi-device version. After analysing the static feature of input code, prediction model will predict the best task partitioning, and the runtime system will take charge of the remaining execution.

However, due to the limitation of training set, the prediction model still need to be improved. Moreover, The framework proposed by Klaus Kofler et al. didn't provide any mechanism to ensure data consistency.

2.3 Conclusion

In this chapter, we have introduced existing standard programming languages and programming tools for heterogeneous architectures. Considering the performance and portability [28, 7, 27] of existing programming language, only OpenACC and OpenCL are supported by many vendors. However OpenACC is not suited to multi-accelerator architectures. Most of existing programming tools are designed for one specific problem which is not convenient enough for novice programmers. In our research, we proposed a programming tool STEPOCL that generates optimised OpenCL code for heterogeneous multi-device architectures. STEPOCL guarantees the portability and performance of OpenCL applications.

Chapter 3

STEPOCL

L'homme qui sait réféchir est celui qui a la force illimitée.

—*Honoré de Balzac*

Contents

In this chapter, we present more details about programming on heterogeneous multi-device architectures in OpenCL. Then, we introduce our contribution STEPOCL, which is an OpenCL based programming tool, along with a new domain specific language designed for simplifying the development of an application for heterogeneous multi-device architectures.

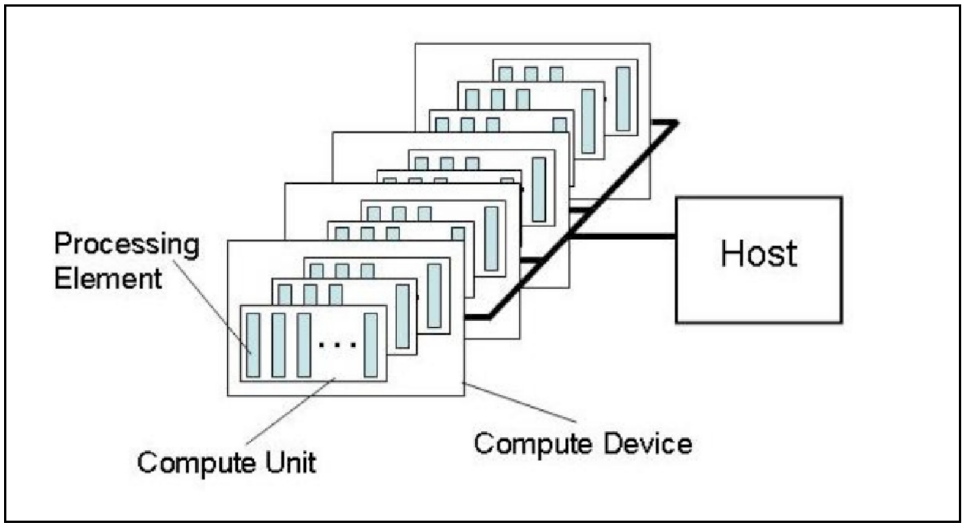


Figure 3.1 – OpenCL Platform Model (image source: KHRONOS group).

3.1 Programming on heterogeneous multi-device architectures in OpenCL

As we presented in Chapter 2, OpenCL provides portability and performance across heterogeneous platforms. It combines a unified programming interface with a variant of the C language to use different parallel processing devices together (e.g. CPU, GPU and Xeon Phi).

3.1.1 Background of OpenCL

Before further investigating programming on multi-device architecture, we briefly introduce some basic concepts of OpenCL.

Platform mode

The platform model of OpenCL consists of a host (a CPU) connected to one or more OpenCL compute devices. A device can be a CPU, a GPU or any other processor supported by the OpenCL vendor. The OpenCL devices are further divided into *compute units* which are further divided into one or more *processing element* (PEs) where the computations on a device occur. Figure 3.1 shows an overview of the OpenCL platform model.

Programming model

OpenCL program consists of two parts: a host code and a computing function. The developer first defines a computing function, called a *kernel*, a basic unit of executable code which is executed on an OpenCL device. In *host code*, developer sets up the environment for the execution of kernel and orchestrates the copy of the data used by the kernel on the device before launching the execution of kernel on a chosen device. When

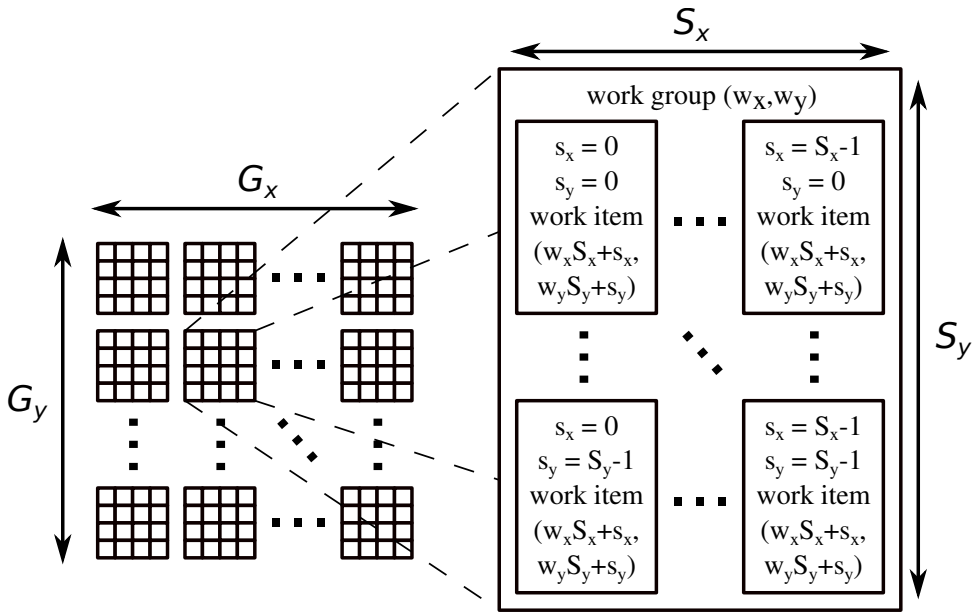


Figure 3.2 – OpenCL Work-Groups and Work-Items.

a kernel is submitted for execution by the host, an index space is defined. Each point in this index space presents an instance of executing kernel. In OpenCL, an instance of executing kernel is called a *work-item*. Work-item is identified by its coordinates in index space. These coordinates are the global ID for work-item which is given by a specific OpenCL primitive (the `get_global_id` primitive). Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-item. In each work-group, work-items are assigned a unique local ID, thus a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID.

Figure 3.2 illustrates this notion of *work-group*: a 2D-kernel index space is split into $Gx \times Gy$ *work-groups*, each of which consists of $Sx \times Sy$ work-items.

OpenCL context and command-queue

OpenCL context defines the environment of kernel's execution. An OpenCL context consists of following components:

- Devices: a set of OpenCL devices to be used by the host.
- Kernels: the computing functions that run on OpenCL devices.
- Program objects: the source code program of kernel.
- Memory objects: the objects that contain data which are visible to all devices within a context.

Command-queue is used for the interactions between the host and the OpenCL devices. Each OpenCL device is attached by a command-queue after the definition of context. With command-queue, host can deliver the commands, such as kernel execution, memory movement and synchronization, through command-queue.

Memory model

The memory model of OpenCL defines four distinct memory regions:

- *Host memory*: Host memory region is visible only to the host. OpenCL defines only how the host memory interacts with OpenCL objects and constructs.
- *Global memory*: This memory region is visible for all work-items.Every work-item can do the read or write action on any element in global memory.
- *Local memory*: This memory region is local to a work-group. The elements in local memory are only shared by the work-items in that work-group. For the local memory, OpenCL provides efficient synchronization mechanisms to guarantee the values seen by a set of work-items in a work-group are consistent.
- *Private memory*: This memory region is private to a work-item. The elements in one work-item's private memory are not visible to other work-items.

The OpenCL device memory and host memory supports global memory. The interaction between host memory and global memory can be achieved by *copying* data (clEnqueueRead/WriteBuffer) or by *mapping* and *unmapping* regions of a memory object (clEnqueueMapBuffer/clEnqueueUnmapObject).

Meanwhile, OpenCL defines four address space qualifiers: **__global**, **__local**, **__constant** and **__private**. These qualifiers, used in variable declaration, specify the region of memory that is used to allocate the object.

- The **__global** address space name is used to indicate that a variable is allocated on global memory.
- The **__local** address space name is used to indicate that a variable will be allocated in local memory, it will be shared by all work-items of a work-group.
- The **__constant** address space name is used to indicate that a variable is allocated on global memory but in read-only mode.
- The other unqualified variables inside a kernel function are **__private**, these variables are only performed **READ/WRITE** by actual work-items.

Parallel models

OpenCL supports two different parallel programming models: data parallelism and task parallelism.

Data-Parallel Programming Model The basic idea of data-parallel programming is operating concurrently on a collection of data elements. When an OpenCL kernel is launched by host code, OpenCL index space automatically maps onto OpenCL memory

objects, which means the identification of each work-item (global ID or local ID) also maps onto memory objects. If the kernel doesn't contain any branch statements, each work-item will execute identical operations on a subset of data items (depends on its global ID). If branch statements exist in a kernel, each work-item still execute the same program, but may accomplish different computing tasks. OpenCL also defines vector instructions and types to support Single Instruction Multiple Data (*SIMD*) models.

Task-Parallel Programming Model OpenCL permits that several submitted kernels can be executed at the same time. In this case, programmers must define and manipulate the execution of concurrent tasks for each computing devices. Because the tasks and the computing capacities of devices vary widely, distributing them so that they all finish at about the same time can be difficult.

In this thesis, we focus on dealing with massive data-parallel tasks which are too large for single computing device's memory. We split large tasks into several small subtasks. Then we use task-parallel programming model to schedule the execution of subtasks. Based on this idea, we developed STEPOCL. It can generate workload partitioning and scheduling modules automatically in C language.

Execution of OpenCL program

Overall, a typical execution of an OpenCL application should be follows the steps below:

- First of all, a CPU host initiates OpenCL initiates OpenCL environment where the contexts, compute devices, program objects and command-queues are defined.

- CPU host defines an N-dimensional index space over some regions of DRAM memory where the input data allocated. Every index of this N-dimensional index space will be a work-item and each work-item executes the same kernel.

- CPU host defines the size of work-group for these work-items. Each work-item in a work-group shares a common local memory. If the size of work-group is undefined, OpenCL implementation will determine itself how to be break the global work-items into appropriate work-group instances, but the performance can be unpredictable.

- The global memory supported by OpenCL devices will load DRAM memory, then OpenCL devices execute each work-groups. On NVIDIA hardware, the multiprocessor will execute 32 threads (32 threads is a "warp group") at once. If the work-group contains more threads than this, they will be serialized.

- When the execution of kernel is finished, the memory object produced from the execution will be copied back to DRAM memory.

3.1.2 Challenge of using multiply compute devices

More compute devices means more parallelism, which leads to better performance. However, developing an application for multi-device architectures is difficult.

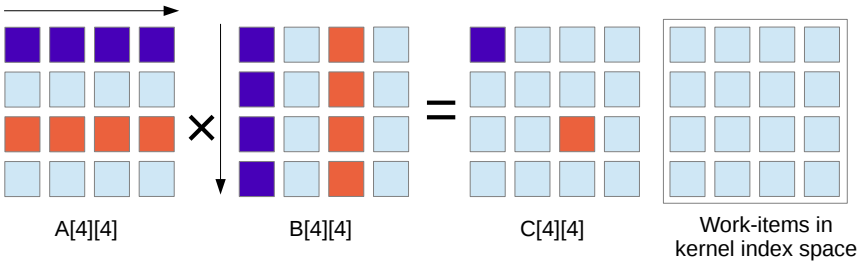


Figure 3.3 – Matrix production on single device

Scheduling is totally hand-made by the programmer

In OpenCL, the process of utilising multiple devices for kernel computation is not done automatically. Programmer should manually controls the process on host code. When using a single device, programmer submits all kernels to the command queue associated with that device. While using multiple devices, programmer has to create a command queue for each device. These command queues either share a same context or each of them has its own context. Then programmer must decide how to distribute the kernels across multiple devices. One of the challenges which has a great impact on the performance of parallel application on heterogeneous multi-device architectures is workload balancing. In order to benefit from the full potential of heterogeneous architecture, programmers are responsible for partitioning and distributing the workload across multiply devices and avoiding the situation in which some devices overloaded while others are underutilized. Manually handling the workload balancing requires that programmers have a solid knowledge of hardware architecture and a rich experience of parallel programming.

Data partitioning

An OpenCL application can contain several kernels. Once these kernels are distributed to different compute devices for the execution, programmers should schedule and synchronize the execution process. If the OpenCL application is data parallel, programmers should manually divide the kernel space and the corresponding data space. Sometimes a precise data partition can be very complicated. Taking a matrix multiplication problem as an example.

Given two matrix $A[n][m]$ and $B[m][p]$, the matrix production $A*B$ is a matrix $C[n][p]$ where the element $C[i][j]$ in matrix$C$ is calculated as below:

$$C[i][j] = \sum_{k=1}^{m} A[i][k] * B[k][j] \tag{3.1}$$

Figure 3.3 illustrates the production of two matrices A and B on single OpenCL device. Each work-item I_{ij} calculates a element $C[i][j]$ through reading all elements of matrix A in row i and all elements of matrix B in column j. All data of matrix A, B and C are allocated on the same physical memory of a OpenCL device. However, if we want to develop a parallel multiply device version, matrix A, B and C should be split into

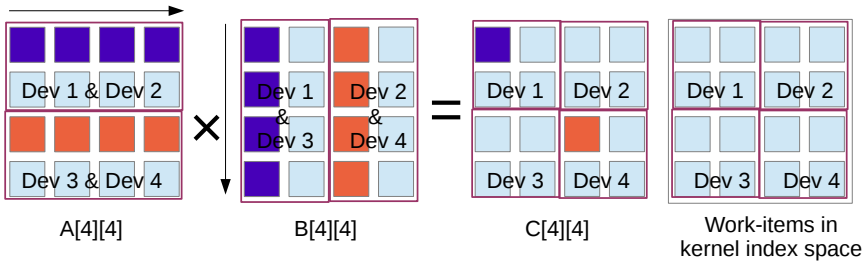


Figure 3.4 – Matrix production on multiple device

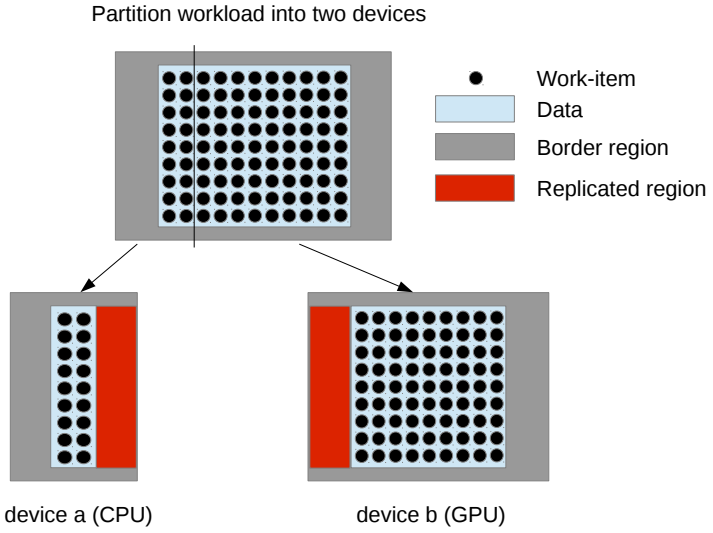


Figure 3.5 – Data consistency

submatrix. Then these submatrix will be allocated on different device memories for the execution of kernel. Figure 3.4 illustrates the workload partition of matrix multiplication problem using four OpenCL devices. The work-items are split into four sets, each set calculates a part of matrix C on one of four OpenCL devices. In order to follow the data dependence, matrix A, B and C are equally split into submatrix which are distributed (or duplicated) to difference devices for further execution. Programmer is responsible for manually estimating the size and position of subregions where the data of submatrix are stored. Due to data dependence, matrix A, B and C are split into different shapes, manually estimating the size and position of these submatrix will become more complicated if more heterogeneous devices are used.

Communication

Another challenge is to maintain data consistency across multiple device memory. Listing 3.1 presents a sequential 2D-stencil iteration. Each update calculates a weighted average of the old cell with its neighbours in the six cardinal direction. In a parallel version for heterogeneous multi-device architecture, the original data will be split into several chunks (the number of chunks is equal to the adequate number of devices following their availability and capabilities.) and be transferred into corresponding dedicated device memory. Figure 3.5 illustrates a possible workload partition for a Jacobi iteration. The data space are split into two subsets. However, due to the data dependencies, two subsets have overlapped areas which is a *replicated region* in Figure 3.5. In order to maintain data consistency across multiple device memory, programmers should have correctly synchronized the preview computation and update the replicated region before each launch of OpenCL kernel for next Jacobi iteration.

However implementing these features is time-consuming and error-prone, there is a huge demand for programming tools that help the novices designing applications for heterogeneous multi-device systems.

Listing 3.1 – Sequential 2D-stencil iteration

```
for (int n = 0, n < t, n++)
{
    for ( int y = 1 ; y < DIM_Y - 1 ; ++y )
        for ( i n t x = 1 ; x < DIM_X - 1 ; ++x ){
            b[ y ][ x ] = alpha * a[ y ][ x ] + beta *
                (    a[ y - 1 ][ x ] + a[ y + 1 ][ x ] +
                     a[ y ][ x - 2 ] + a[ y ][ x + 2 ] ) ;
        }
    swap(a,b);
 }
```

3.2 Component of STEPOCL

STEPOCL programming tool is created for facilitating the development of OpenCL applications. It provides an environment for developing efficient and portable parallel applications on multiple heterogeneous OpenCL devices. Programmer only need to focus on developing the OpenCL computing kernel and let STEPOCL to manage data partition, workload balancing and data consistency.

As illustrated in Figure 3.6, the core component of STEPOCL is a *code generator*, which takes as input a list of raw OpenCL kernels (see Subsection 3.2.1) together with a configuration file (see Subsection 3.2.2) which describes:

- the layout of the data, which expresses how the data shall be split among the devices,
- the association between a compute kernel and a specific device type (see Subsection 3.2.2), and
- the expected control flow of the application to generate.

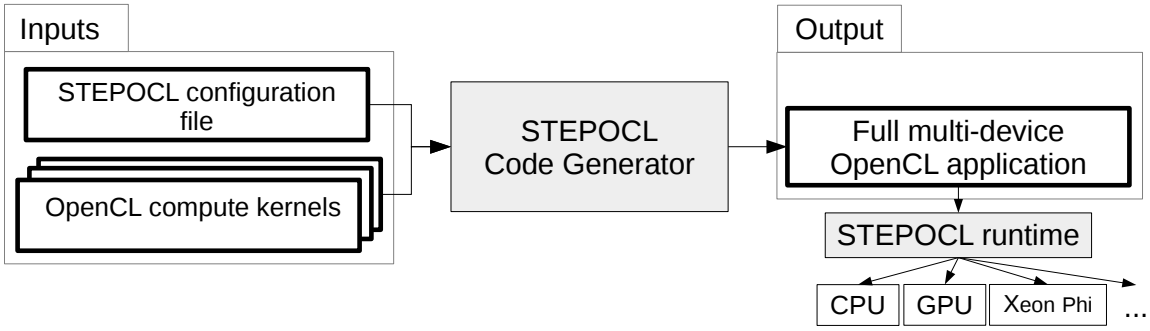


Figure 3.6 – Overview of the STEPOCL environment.

Based on this input, STEPOCL then generates a complete OpenCL program (see Subsection 3.2.3) able to exploit concurrently different accelerators, e.g., CPU, GPU and Xeon Phi that runs on top of the STEPOCL *runtime*.

At bootstrap, the generated code sets up a unified OpenCL multi-device environment and distributes the workload among the accelerators based on the results of the offline profiler. Then, during the run, the generated code maintains data consistency between the devices by using the result of a data flow analysis performed during compilation.

3.2.1 Compute kernels

The input kernels are regular OpenCL kernels, which express the computation to execute on a device. As a basic example for the remaining of this Section, we provide a 1D-stencil kernel written for a single generic device in Listing 3.2.

Listing 3.2 – Generic OpenCL 1D-stencil kernel.

```
__kernel void generic_stencil1D(__global float *A, __global float *B) {
    const unsigned int i = get_global_id(0) + 1;
    B[i]= (A[i-1]+A[i]+A[i+1])/3;
}
```

The developer may prepare several versions of the compute kernel to achieve the best performance on a specific device type. For instance, in order to favour data cache effects of a CPU in the 1D-stencil, it is more efficient to process data by tile, *i.e.* by block of elements instead of element-by-element, as presented in Listing 3.3. In this case, the computation performed by the CPU remains the same, but the amount of work-items differs.

Listing 3.3 – Tiled OpenCL 1D-stencil kernel.

```
__kernel void tiled_stencil1D(__global float *A, __global float *B) {
    const unsigned int i = get_global_id(0) + 1;
    for(int k = 0; k < 4; k ++)
    B[i+k]= (A[i+k-1]+A[i+k]+A[i+k+1])/3;
}
```

3.2.2 STEPOCL configuration file

The configuration file takes the form of a tree, implemented in XML format. It defines the *arguments* of the kernel, how the *kernels* are mapped to the device type, and the *control flow* of the program.

`Argument`

An `argument` describes an array of values, transferred to the devices and accessed by the work-items. It is defined by three elements:

- A name (`ID`), later used in the kernel section to refer to the argument.
- The type (`data_type`) of its tokens.
- Its `size`, which further specifies the dimension and length of the array.

For instance, the configuration file given in Listing 3.4 defines the arguments of our 1D-stencil example: arrays `A` and `B` are two 1D vectors, which contain 1026 float values each.

Listing 3.4 – Argument description of the 1D-stencil kernel.

```
<argument>
  <ID> A </ID>
  <data_type> float </data_type>
  <arg_size>
    <dim_size axis=x> 1026 </dim_size>
  </arg_size>
</argument>
<argument>
  <ID> B </ID>
  <data_type> float </data_type>
  <arg_size>
    <dim_size axis=x> 1026 </dim_size>
  </arg_size>
</argument>
```

`Kernel`

The `kernel` section of the configuration file describes the mapping between the kernels and the device types. It contains three elements:

- A `name` element. It references the kernel in the subsequent control flow section.
- A `data_split` element. For each argument, it gives along which dimension its related data should be split.
- At least one `implem` element. An implem element associates a compute kernel code to a specific device and defines the size of a *tile*, *i.e.* the size of the data to compute by each work-item. The default tile size is 1.

The kernel section may contain several *implem* elements in order to associate different kernel codes (`funcname`) to different devices (`device_type`). The DEFAULT `implem` is used when no other `implem` corresponds to the device.

As introduced in Section 2.2.2, the programmer can also specify the size of the work-groups (`work-group`) used to design a specific kernel version.

Listing 3.5 presents the kernel configuration of the 1D-stencil. The user expresses that arrays `A` and `B` shall be split by column (*i.e.* split along the `x` axis). Three versions of the kernel are provided. The first version (the `tiled_stencil1D` kernel) targets CPU devices and performs its computation on tiles of four elements. The second version (the `GPU_stencil1D` kernel) is designed for GPU devices. It specifies the aggregation of work-items by groups of 16 along the x axis, which means that a group of 16 work-items can share their memory. The third kernel version (the `generic_stencil1D` kernel) is the more basic as it uses neither tile nor work-group. This version is also the more generic as it can be used on any other available device.

Listing 3.5 – 1D-stencil kernel information.

```
<kernel>
  <name> Stencil1D </name>
  <data_split>
    <axis ID=B> x </axis>
    <axis ID=A> x </axis>
  </data_split>
  <implem>
    <platform>Intel(R) OpenCL</platform>
    <device_type> CPU </device_type>
    <funcname> tiled_stencil1D </funcname>
    <tile>
      <target_arg> B </target_arg>
      <size axis=x> 4 </size>
    </tile>
  </implem>
  <implem>
    <platform>NVIDIA CUDA</platform>
    <device_type> GPU </device_type>
    <funcname> GPU_stencil1D </funcname>
    <work_group>
       <size axis=x> 16 </size>
    </work_group>
  </implem>
  <implem>
    <platform>Intel(R) OpenCL</platform>
    <device_type> ACCELERATOR </device_type>
    <funcname> Acc_stencil1D </funcname>
  </implem>
</kernel>
```

`Control`

The third component of the STEPOCL configuration file describes the program `control` flow. The control flow is basically the meta-algorithm of the application and is used by the data analysis pass to ensure consistency. This component describes the number of iterations of the kernel to launch (`loop` keyword) and how data are exchanged between two iterations (`switch` keyword). For instance, in Listing 3.6, the STEPOCL application executes 10 times the `Stencil1D` and has to switch the arguments `A` and `B` between two iterations.

Listing 3.6 – 1D-stencil kernel execution.

```
<control>
  <loop iterations=10 >
    <exec> Stencil1D </exec>
    <switch>
      <arg ID=A> B </arg>
    </switch>
  </loop>
</control>
```

3.2.3 STEPOCL Output of OpenCL application

Based on the kernel(s) and on the configuration file, STEPOCL generates a multi-device OpenCL host program. The generated codes are saved in two files *run.c* and it's header file *run.h*. *run.c* predefines the modules of initialisation of OpenCL environment, data partition, communication and scheduling. Function *func* in file *run.c* is the core program which defines the execution process of host code.

Listing 3.7 – run.c of 1D-stencil code

```
#include "run.h"
/***** predefinition of functions ******/
    .... ....
/***** host code ******/
void func(float* h_idata, float* h_odata){
    .... ....
}
```

Listing 3.7 and listing 3.8 presents the method of using generated code. We take still 1D-stencil computation as an example. Once STEPOCL generated the host code, the only thing left for programmer is to initialize the data of input in *main.c* file. In listing 3.8, we initialized two arrays *h_idata* and *h_odata*, then we called function *func* for the 1D-stencil computation. According to predefined configuration file, *func* will manage automatically all technical aspect of parallel programming problems that we mention in section 3.1.2.

Listing 3.8 – main.c of 1D-stencil code

```
#include "run.h"

#define XDIM 1000002
#define LINESIZE   XDIM
#define TOTALSIZE  XDIM

int main(int argc,char* argv[]){
    float* h_idata = NULL ;
    float* h_odata = NULL ;

    const unsigned int mem_size = TOTALSIZE*sizeof(float);
    h_idata = (float *)malloc(mem_size);
    h_odata = (float *)malloc(mem_size);

    for(unsigned int i = 0; i < TOTALSIZE; i++){
        h_idata[i]=i;
        h_odata[i]=0.0;
    }

    func(h_idata,h_odata);
    return 0;
}
```

3.3 STEPOCL internal mechanism

3.3.1 Automatic device management

In a regular OpenCL program, programmer is responsible for manually initiating OpenCL environment, choosing available device, creating context and then associating kernels and command queues for each device. STEPOCL automatically generates these parts. According to the specification described in Figure 3.5, the generated host code searches devices which matches the specified requirement from different OpenCL platforms. On an OpenCL platform, host code creates a context for each type of compute devices. Then each context associates an OpenCL kernel and each device is associated to a command queue. Figure 3.7 shows the concept of device management in OpenCL environment. STEPOCL user is also able to use one context for all devices on the same platform. For example, in listing 3.9, the *device_type* defined under the *platform* is **ALL**. Thus STEPOCL will create a context for all devices on the same platform, each devices run the same kernel(Figure 3.8).

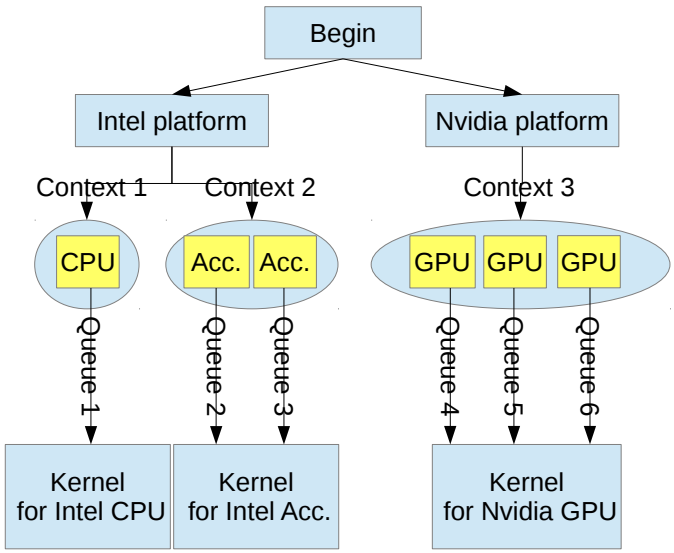


Figure 3.7 – Device management: using different kernel

Listing 3.9 – 1D-stencil kernel information.

```
<kernel>
  <name> Stencil1D </name>
  <data_split>
    <axis ID=B> x </axis>
    <axis ID=A> x </axis>
  </data_split>
  <implem>
    <platform>NVIDIA CUDA</platform>
    <device_type> ALL </device_type>
    <funcname> stencil </funcname>
    <work_group>
```

```
        <size axis=x> 16 </size>
      </work_group>
    </implem>
    <implem>
      <platform>Intel(R) OpenCL</platform>
      <device_type> ALL </device_type>
      <funcname> stencil </funcname>
    </implem>
  </kernel>
```

3.3.2 Automatic workload partition

STEPOCL runtime evaluates the performance of selected compute devices, and then automatically performs the workload partition. STEPOCL profiler (see Section 4.2) collects the static data of compute devices with OpenCL API, then each selected device is associated with a ratio which represents its relative performance capacity on whole system.

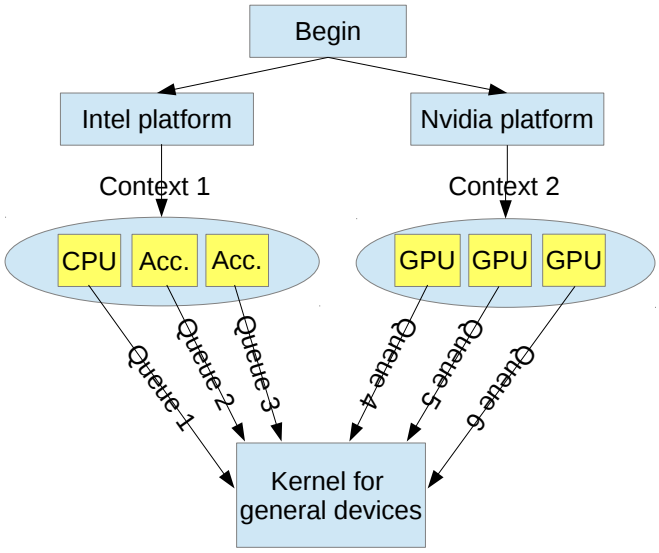


Figure 3.8 – Device management: using a common kernel

Based on the ratio and on the tile size (provided by the configuration file, see Section 3.2.2) of each device, the generated code computes the proportion of workload for each device, then partitions kernel space and data space (see Section 4.3) with corresponding proportion. Meanwhile, the compiler of STEPOCL (see Section 4.1.1) launches region analysis on OpenCL kernel, the analyse result is used for data partitioning and managing the data consistency.

3.3.3 Automatic adjusting workload

Iterative applications, which are typical in HPC, relaunch kernels until the result meets the requirement. For example, in each time step n of Jacobi iteration in listing 3.1, OpenCL kernel will be executed on each selected compute device. STEPOCL runtime keeps watching over the status of execution on each device, then it adjusts the workload (if necessary) to keep the load balancing on whole systems (see Section 4.4).

3.3.4 Automatic data consistency management

In OpenCL, data consistency on global shared memory can be achieved with *barrier* or $fence$. However, this mechanism doesn't work across multiple devices. Programmers have to manually synchronize the execution of kernel using command queues, then update intermediate results on dedicated memory of each device. With the information of region analysis and the proportion of data partition, STEPOCL estimates the size and position of regions where the data need to be updated. Then, STEPOCL automatically generates the module of kernel synchronization and the module of communication for intermediate result update (see Section 4.5).

3.4 Conclusion

In this chapter, we have introduced STEPOCL which is our answer to the existing problems of programming on heterogeneous architectures. STEPOCL automatically generates the host code of OpenCL, programmers only need to provide compute kernels and a configuration file where the information of input data and execution are specified. The elements and attributes defined in configuration file are familiar to OpenCL programmer since STEPOCL uses the same concepts of OpenCL objects. The generated host code is capable of performing workload partition, workload balancing and management of data consistency on any multiple devices heterogeneous architectures.

Chapter 4

Implementation

Plus tu travailles dur, plus tu es chanceux.

—*Thomas Fuller*

Contents

This chapter presents the main modules implemented in STEPOCL including analysis of regions, offline profiling, workload partition, workload balancing, communication. The results of region analysis and the information of offline profiling are used for workload partitioning and the management of communication. The module of workload balancing is implemented for monitoring the status of each computing devices, STEPOCL adjusts the proportion of workload on each device if the unbalanced status is detected. The structure of generated code are presented at the end of chapter.

4.1 Analysis of region

Workload partitioning requires precise array data flow analysis. In this section, we concisely introduce compiler PIPS [16], a compiler, and how to analyse array data flow that allows us to generate them.

INPUTS	C	Fortran 77	
ANALYSES	Array Privatization	Array Section Privatization	Array Element Region
	Control Flow Graph	Continuation Conditions	Dependences
	Preconditions	Program Complexity	Reduction Detection
	Transformers	Use-Def Chains	Call Graph
	Memory Effects	Scalar variable Privatization	
RESTRUCTURATIONS	Atomization	Cloning	Control Restructuration
	Useless Definition Elimination	Declaration Cleaning	Dead Code Elimination
OUTPUTS	Call Graph	Control Flow Graph	Dependence Graph
	Parallel Fortran 77	C	Optimised Fortran 77
TRANSFORMATIONS	Coarse Grain Parallelization	Expression Optimizations	Forward Substitution
	Loop Distribution	Loop Interchange	Loop Normalize
	Loop Reductions	Loop Unrolling	Partial Evaluation

Table 4.1 – PIPS overview

4.1.1 PIPS

PIPS is a source-to-source compilation framework. It takes as input Fortran or C codes and uses inter-procedural techniques for program analyses such as precondition [15] and array region [9, 23] computation. PIPS can also perform code transformation such as loop interchange, tiling or can even generate parallel Fortran or C code. Table 4.1 lists the modules and functions of PIPS.

PIPS provides a line interface *tpips* to PIPS users. As a scripting language of the PIPS project, *tpips* can access to Unix shell and to PIPS properties.

4.1.2 Analysis of regions with PIPS

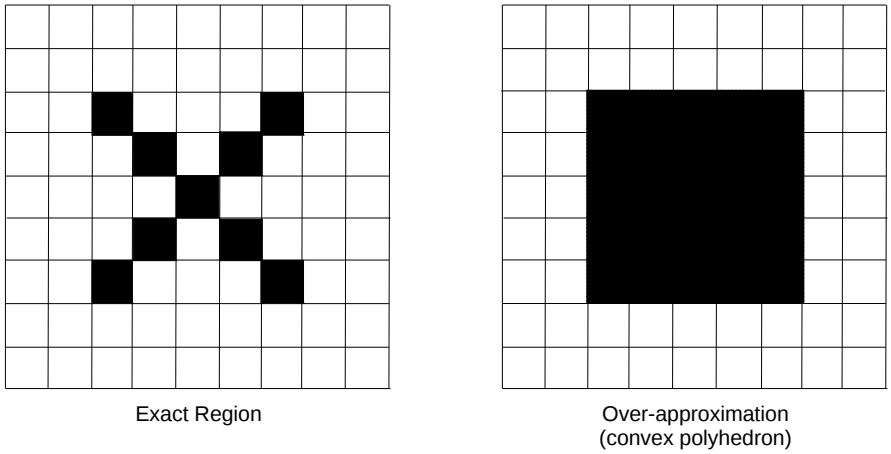


Figure 4.1 – Over-approximated analysis

In PIPS, array regions are represented by convex polyhedra [8]. They are used to summarise accesses to array elements. Due to region representation, the analysis can be over-approximated as Figure 4.1 presented. In PIPS, *EXACT* region represents exactly

matched array region, and *MAY* region represents the array region which can be over-approximated. The *READ* and *WRITE* regions represent the effects of statements and procedures on sets of array elements. PIPS also provides *IN* and *OUT* regions analysis to represent array data flow. For a block of statements or a procedure, an *IN* region is the subset of the corresponding *READ* region containing the array elements that are imported and an *OUT* region is the subset of the corresponding *WRITE* region that will be read outside of block.

Program example for array regions analysis. The program given in the Listing 4.1 is a C program which doubles the value of elements in a one-dimensional array. The *Main* function initializes a one-dimension array with ten elements, then calls a *loop* to update the elements in the array.

Listing 4.1 – C program for array regions analysis

```
#include <stdio.h>
#define N 10

void loop(int l, int u,int array[N])
{
  int i;

  for(i=l; i<u-1; i++)
  {
    array[i]=2*i;
  }
}

int main(void)
{
  int array[N];
  int i,j;

  for(i=0; i<N; i++)
  {
    array[i]=i;
  }

  loop(0,N,array);
  return 0;
}
```

Array region analysis on this example. Based on the program on Listing 4.1, PIPS analyses array region at every statement level of the abstract syntax tree. For example, an analysis result inside the *loop* function can be seen in Listing 4.2.

Listing 4.2 – Array regions analysis by PIPS

```
#include <stdio.h>
#define N 10

// <array[PHI1]-W-MAY-{0<=PHI1, PHI1<=9, l==0, u==10}>
// <array[PHI1]-OUT-MAY-{0<=PHI1, PHI1<=9, l==0, u==10}>
void loop(int l, int u,int array[N])
{
  int i;

// <array[PHI1]-W-MAY-{0<=PHI1, PHI1<=9, l==0, u==10}>
// <array[PHI1]-OUT-MAY-{0<=PHI1, PHI1<=9, l==0, u==10}>
  for(i=l; i<u; i++)
  {

// <array[PHI1]-W-EXACT-{PHI1==i, l==0, u==10, 0<=i, i<=9}>
// <array[PHI1]-OUT-MAY-{PHI1==i, l==0, u==10, 0<=i, i<=9}>
    array[i]=2*i;
```

```
    }
}

int main(void)
{
  int array[N];
  int i,j;

//  <array[PHI1]-W-EXACT-{0<=PHI1, PHI1<=9}>

  for(i=0; i<N; i++)
  {
//  <array[PHI1]-W-EXACT-{PHI1==i, 0<=i, i<=9}>
    array[i]=i;
  }

  loop(0,N,array);
  return 0;
}
```

In Listing 4.3, *PHI1* represents the index of array *array*. The *WRITE* region (represented by W) of *array* is exactly the sub-array *array[PHI1]* with: $PHI1 == i, 0 \leq i, i \leq 9$. The symbol OUT means that *array* is updated in current block or statement, it might be transmitted or updated in later.

Listing 4.3 – Array regions analysis by PIPS

```
//  <array[PHI1]-W-EXACT-{PHI1==i, l==0, u==10, 0<=i, i<=9}>
//  <array[PHI1]-OUT-MAY-{PHI1==i, l==0, u==10, 0<=i, i<=9}>
```

The analysis of *READ*, *WRITE*, *IN* and *OUT* provide the information of data dependencies and the size of each region. In our work, we use PIPS to perform region analysis on OpenCL kernels, because we require the information of region size to partition the workload, we also require the information of data dependencies to perform data exchanges.

4.1.3 Analysis of OpenCL kernel

In the scope of STEPOCL, OpenCL kernels are written in OpenCL C programming language which is based on the ISO/IEC 9899:1999C language specification (also known as C99 specification) with specific extensions and restrictions. Some features of OpenCL C are not supported by the PIPS compiler, thus some data types or functions should be redefined before passing region analysis with PIPS.

OpenCL type for application	Corresponding C type
cl_char	char
cl_uchar	unsigned char
cl_short	short
cl_int	int
cl_uint	unsigned int
cl_long	long
cl_ulong	unsigned long
cl_float	float
cl_double	double
cl_half	half

Table 4.2 – OpenCL scalar data type

Table 4.2 describes the list of OpenCL scalar data types that should be converted before using PIPS. OpenCL also provides vector data types. They are defined with the type name (char, int and float etc.) followed by a number n that defines the number of elements in the vector. These vector data types presented in Table 4.3 should be converted into corresponding scalar data types to adapt PIPS's C parser.

OpenCL type for application	Description
cl_char$n$	A vector of n 8-bit signed two's complement integer values.
cl_uchar$n$	A vector of n 8-bit unsigned integer values
cl_short$n$	A vector of n 16-bit signed two's complement integer values.
cl_ushort$n$	A vector of n 16-bit unsigned integer values.
cl_int$n$	A vector of n 32-bit signed two's complement integer values.
cl_uint$n$	A vector of n 32-bit unsigned integer values.
cl_long$n$	A vector of n 64-bit signed two's complement integer values.
cl_ulong$n$	A vector of n 64-bit unsigned integer values.
cl_float$n$	A vector of n 32-bit floating-point values.
cl_double$n$	A vector of n 64-bit floating-point values.

Table 4.3 – OpenCL vector data type

The address space qualifiers (in Section 3.1.1) and built-in work-item functions are redefined beside PIPS for region analysis at single work-item level. Listing 4.4 and Listing 4.5 present a classic 1D-stencil kernel and its result of region analysis. Each work-item performs three *READ* actions on table A and one *WRITE* action on Table B. For each work-item, the relative interval of *READ* action on table A is $[-1, 1]$ in one dimension and the interval of *WRITE* action on table B is [0,0]. Meanwhile, the interval of *READ* action on table A presents the range of ghost region. When the large region is split into subregions, each subregion shall keep the extra ghost region for computational purpose. In some applications, such as stencil iteration computation, the ghost regions (as intermediate results) need to be updated from other subregion after each iteration. Thanks to the information of data dependency provided by region analysis, STEPOCL can manage automatically the communication of intermediate result.

Listing 4.4 – 1D-stencil kernel

```
__kernel void generic_stencil1D(__global float* A, __global float* B)
{
      const unsigned int i = get_global_id(0)+1;
      A = A+1;
      B = B+1;
      B[i] = (A[i-1]+A[i]+A[i+1])/3;
}
```

Listing 4.5 – Analysis of 1D-stencil kernel

```
__kernel void generic_stencil1D(__global float* A, __global float* B)
{
      const unsigned int i = get_global_id(0)+1;
      A = A+1;
      B = B+1;
  //  <A[PHI1]-R-EXACT-{PHI1==i+1, i==0}>
  //  <A[PHI1]-R-EXACT-{PHI1==i, i==0}>
  //  <A[PHI1]-R-EXACT-{PHI1==i-1, i==0}>
  //  <B[PHI1]-W-EXACT-{PHI1==i, i==0}>

      B[i] = (A[i-1]+A[i]+A[i+1])/3;
}
```

cl_device_info	Description
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	Return type: cl_ulong Size of global memory cache in bytes.
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	Return type:cl_device_mem_cache_type Type of global memory cache supported.
CL_DEVICE_MAX_CLOCK_FREQUENCY	Return type: cl_uint Maximum configured clock frequency of the device in MHz.
CL_DEVICE_MAX_COMPUTE_UNITS	Return type: cl_uint The number of parallel compute cores on the OpenCL device. The minimum value is 1.
CL_DEVICE_NAME	Return type: char[] Device name string.
CL_DEVICE_PLATFORM	Return type: cl_platform_id The platform associated with this device.

Table 4.4 – Get information about an OpenCL device

4.2 Offline profiling

In order to ensure the load balancing on heterogeneous architectures, it is important to have the information about the problem size and the hardware configurations. STEPOCL offline profiling is used for evaluating the performance of OpenCL devices.

Modern multi-core processors are very complex architectures. There are many factors that affect a processor performance and the performance model varies among different processor types. For example, CPU has less latency issues than GPU. Some factors, such as the impact of cache architecture and the register size, are difficult to analyse. Meanwhile, the input data size also affects the efficiency of performance.

In order to simplify the problem, STEPOCL assumes that developer want to use the full potential of whole heterogeneous platforms, which means that the input data set is large enough to saturate all computing devices. The performance model used by STEPOCL is expressed in a simple form as Equation 4.1.

$$Performance = CoreCount * Frequency * \frac{InstructionPerCycle}{InstructionCount} \tag{4.1}$$

Considering that OpenCL applications are massively parallel, the *Instruction per cycle* for each core should be 1. Meanwhile, the instruction count should be equal for all compute devices, since they execute the same kernel. Thus, there are only *Core count* and *frequency* that we shall consider. OpenCL API *clGetDeviceInfo* provides the functionality that permits programmers to access the critical attributes of an available compute device. Table 4.4 lists a part of information that can be accessed through *clGetDeviceInfo* function.

STEPOCL assesses the performance of each device i ($perf_i$) during the initialization of OpenCL environment. The performance of whole heterogeneous system can be expressed as the sum of each $perf_i$ (in Equation 4.2, n is the number of available compute devices).

$$perf\_system = \sum_{i=0}^{n} perf_i \tag{4.2}$$

At the moment of workload partition, the proportion of workload for each device $prop_i$

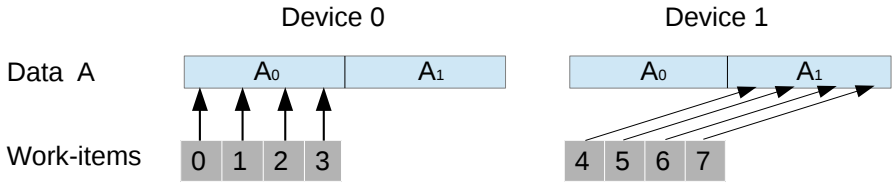


Figure 4.2 – Workload partitioning for multi-device platform with single context

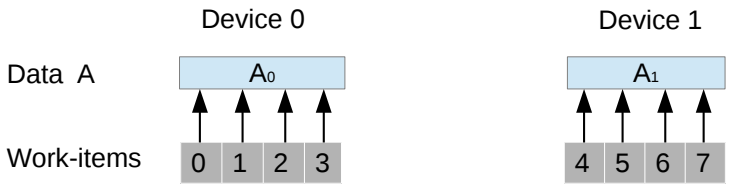


Figure 4.3 – Workload partitioning for multi-device platform with multiply contexts

is calculated in Equation 4.3 .

$$prop_i = \frac{perf_i}{perf\_system} \tag{4.3}$$

4.3 Workload partition

There are two methods for partitioning an OpenCL workload:

- Redundantly copy all data to each computing device and index the work-items using global offsets (Figure 4.2).
- Split the data into subsets and index the work-items into the subset (Figure 4.3).

The first method uses only one OpenCL context for all computing devices. The benefit of this method is that it is an easy way to manage data, developers do not have to manually code the data partition for each device. However the shortcomings are also obvious. This method can not work on a mixed platform (e.g. combining an Intel CPU with Nvidia GPUs and AMD GPUs). Meanwhile, redundantly copying all data to each computing device consumes a lot of memory resources. The memory available on each computing device can be different, some devices may not even have enough space to allocate the whole data.

The second method is to create redundant OpenCL contexts for computing devices. This method can directly access all computing devices on different OpenCL platforms.

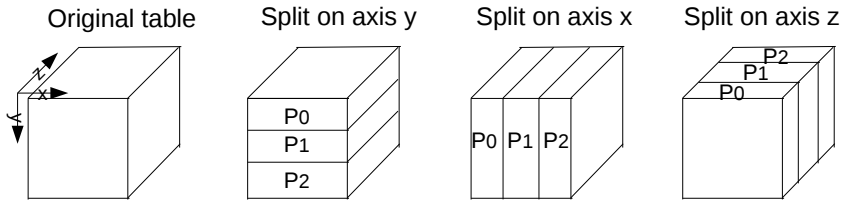


Figure 4.4 – Available data partitioning over multiple devices

The utilization of device's memory is also more efficient than the first method, Since no redundant data is allocated on the device. STEPOCL adopts the second method, and automatically generates the code of data partition.

4.3.1 Data space partition

The data space is partitioned with the pre-estimated proportion thanks to the profiler (Section 4.2), however the details of partition depends on the description of STEPOCL configuration file. Figure 4.4 presents the possible partitions of a 3D data for three computing devices.

Ghost region

For some applications, updating an element of data region requires neighbouring region. In order to localize the computation, the subregions that are split from original region should contain the replicated neighbouring regions (called *ghost* region). Figure 4.5 presents the data partition of a 2D table for four computing devices with ghost region.

The size of ghost region can be calculated from the result of region analysis (Section 4.1). The relative interval (α_0, α_1) of $READ$ action of a single work-item represents exactly the border information of ghost region.

Partitioning process

Assuming that we have N devices, the proportion of performance for each devices P_i is $prop_i$, the projection of ghost region on axis $axis_n$ is (α_0, α_1), the data size on $axis_n$ is $Dsize$, the tile size on $axis_n$ for each device P_i is $tile_i$ and the size of work-group on $axis_n$ for device P_i is wg_i. Then the data partition on $axis_n$ will be achieved in following steps:

First of all, the data used for computation $Dsize\_c$ is computed as below.

$$Dsize\_c = Dsize - \alpha_0 - \alpha_1 \tag{4.4}$$

According to the statistics of device performance, $DataP_0$ (the preallocated data for device P_0) is calculated in equation 4.5

$$DataP_0 = \lfloor Dsize\_c * prop_0 \rfloor \tag{4.5}$$

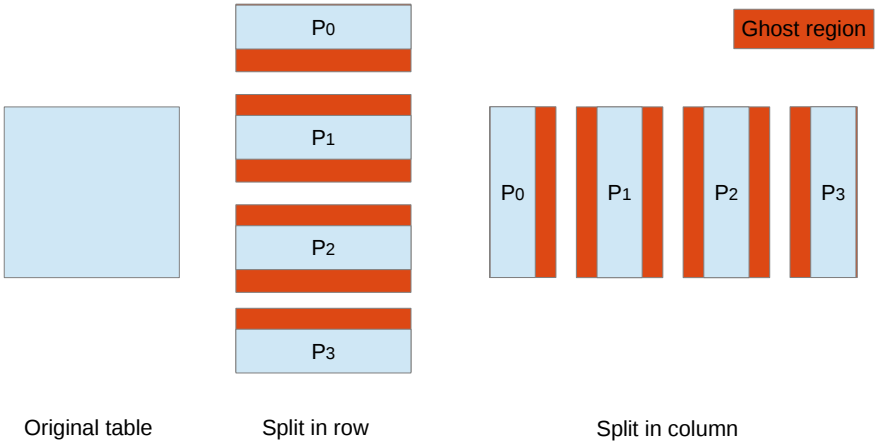


Figure 4.5 – Data partition with ghost region

Next, adjusting the data size $DataP_0$ to match the tile size. The tile size described in STEPOCL configuration file represents the size of data to compute by each work-item. We must ensure that each work-item can access to its own data region. Thus the size of data must be divisible by the tile size. Equation 4.6 presents the process of adjusting $DataP'_0$.

$$DataP'_0 = DataP_0 - DataP_0\%(tile_0 * gw_i) \tag{4.6}$$

Then the size of data that will be allocated on device P_0 is $DataP'_0$ with its ghost region (Equation 4.7).

$$DataOnP_0\_Axis_n = \alpha_0 + DataP'_0 + \alpha_1 \tag{4.7}$$

Finally, the data size for the remaining devices can be calculated step by step in the same way as $DataOnP_0\_Axis_n$. However if the data space is a multi-dimensional table, the final data size for device t will be calculated as Equation 4.8. In Equation 4.8, n represents the number of dimension. Figure 4.6 illustrates a complete partition process on one dimension.

$$DataOnP_t = \prod_{i=0}^{n-1} DataOnP_t\_Axis_i \tag{4.8}$$

After data partitioning, each subregion contains the following information: the global **ID**, the information of written data region (*data_region*) and the information of ghost region (*ghost_region*). The global **ID** of subregion can be represented with relative ID (re_ID_x, re_ID_y) and device partitioning information (the number of devices in the first dimension *numdev_x*, the number of devices in the second dimension *numdev_y*).

$$\begin{aligned} global\_ID \quad = \quad & re\_ID\_y \times numdev\_y + \\ & re\_ID\_x(2D : global\_ID) \end{aligned} \tag{4.9}$$

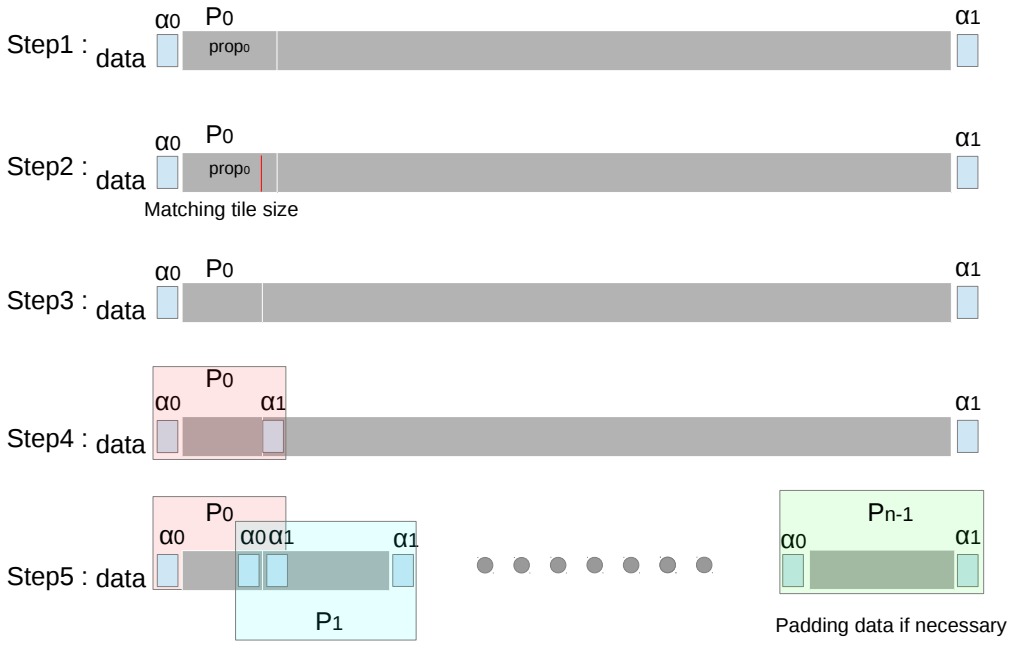


Figure 4.6 – The process of Data partition

$$\begin{aligned} global\_ID &= re\_ID\_z \times numdev\_y \times numdev\_x + \\ &\quad re\_ID\_y \times numdev\_y + \\ &\quad re\_ID\_x(3D : global\_ID) \end{aligned} \tag{4.10}$$

STEPOCL uses these **ID**s to locate the neighbouring **ID**s for each subregion and keeps these neighbouring information in a data structure called **neighbour_list**.

4.4 Workload balancing

This section presents the mechanisms that are used by STEPOCL to dynamically adjust the workload of each computing devices. The first workload partitioning may be not perfectly balanced. STEPOCL follows the status of several executions at the beginning, then it will adjust the proportion of workload if unbalances are detected.

In order to distinguish data ratio from performance ratio $prop_i$, we use r_i^0 to represent the initial data ratio that has to be deployed on the device i at the first time of workload partition.

Once the computing kernels are executed, STEPOCL runtime observes the elapsed time d_i^j to perform the iteration j on each device i. After each iteration, STEPOCL runtime copies all the output arguments that have to be switched with an input argument in the main memory, re-estimates a new ratio r_i^{j+1} based on the elapsed time and the previous ratio r_i^j, and redeploys the input arguments on the device.

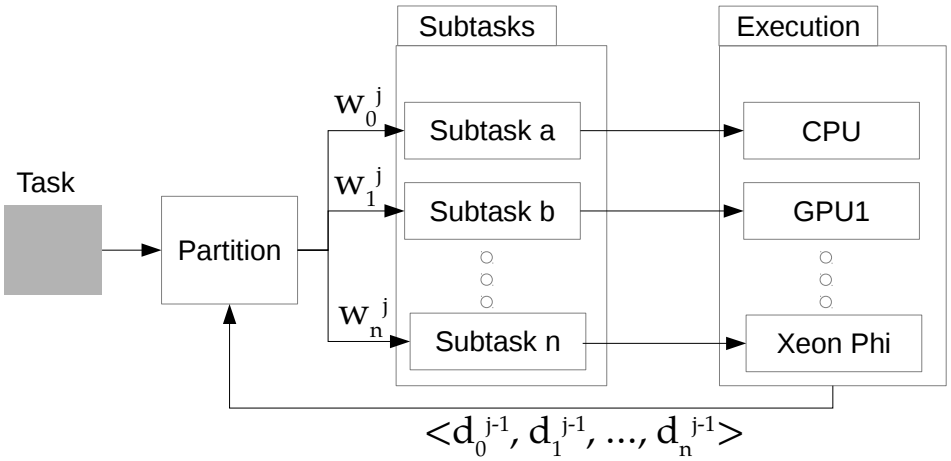


Figure 4.7 – Overview of the load balancing algorithm used in the STEPOCL runtime.

Figure 4.7 summarizes the mechanism. In the Figure, w_i^j represents the workload of the device i at the iteration j, *i.e.*, the size of the input data multiplied by r_i^j.

A naive way to adjust the ratios after each iteration would be to consider the ratio between $\overline{d^j}$, the mean duration at iteration j, and d_i^j, for instance, by defining r_i^{j+1} as $r_i^j \times 1 + \overline{d^j}/d_i^j$. The main drawback we may encounter by scheduling with this method is the occurrence of the ping-pong effect illustrated in Figure 4.8. Let us consider two homogeneous devices A and B. Let us assume, for instance, that due to the instability caused by cache misses or some other reasons, the evaluation of the duration of iteration j for CPU_a is over estimated. Then at iteration $j+1$, the naive re-balancing formulae will assign more work-items to CPU_b than to CPU_a. As a result, at the iteration j, the offline profiler considers that CPU_b works slowly and underloads it. We have observed that the naive re-balancing formulae can overload CPU_a at iteration $j+2$ and that this phenomenon may be amplified.

In order to reduce the risk of this ping-pong effect, STEPOCL adapts the changing speed of the ratio adjustment from one iteration to another. The main idea behind this method is to reduce this changing speed whenever an inversion of the direction of variation of r_i^j is detected. To this end, a value Q, initialized to 1, is incremented after each such

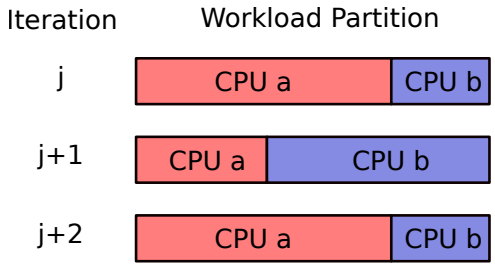


Figure 4.8 – Ping-pong effect of workload adjustment.

inversion.

We consider that when Q increases, *i.e.*, that after few inversions, the re-balancing factor $\overline{d^j}/d_i^j$ should become less important because the workloads are converging to an efficient configuration. For this reason, we define the ratio for iteration j and for device i as follows:

$$r_i^{j+1} := r_i^j \left(1 + \frac{1}{Q} \times \left(\frac{\overline{d^j}}{d_i^j} - 1\right)\right);$$

The self-adjustment process keeps on running until σ^j, defined as the standard deviation of execution time on each device for iteration j, becomes small enough. In STEPOCL, we consider that the workload is "calibrated" when $\sigma^j < 0.05\ \overline{d^j}$. Once the calibration is complete, the new ratios r_i^{j+1} are returned, so that the next invocations of the generated application distribute the workload efficiently among the tested devices.

4.5 Data transmission between multiple devices

This Section introduces how STEPOCL manages data transmission between multiple devices.

Read/write memory objects

STEPOCL uses buffer objects for the movement of data in and out of the compute device memory, from the host's memory. These memory objects are stored in the host memory (typically, in RAM) or in the device memory (typically, in GRAM directly on the graphic card). There are several functions that can be used to read and write memory object. The Table 4.5 presents five functions that read and write buffer objects.

Function	Purpose
clEnqueueReadBuffer	Reads data from a buffer object to host memory
clEnqueueWriteBuffer	Writes data from host memory to a buffer object
clEnqueueReadBufferRect	Reads a rectangular portion of data from a buffer object to host memory
clEnqueueWriteBufferRect	Writes a rectangular portion of data from host memory to a buffer object
clEnqueueCopyBuffer	Enqueues a command to copy a buffer object to another buffer object

Table 4.5 – read and write buffer objects

Exchanging the intermediate results between devices

The intermediate data that needs to be transferred from device **A** to device **B** can be calculated as the intersection of the region that is written on device **A** and will be read

on device **B** (see equation 4.11).

$$region\_aTob = exact\_write\_A \cap exact\_read\_B \tag{4.11}$$

Since STEPOCL defines *data_region* as EXACT_WRITE region and *ghost_region* as EXACT_READ region, the equation 4.11 can be redefined as:

$$region\_aTob = data\_region\_A \cap ghost\_region\_B \tag{4.12}$$

Then all the data can be transferred with the **neighbor_list** following the process presented in Listing 4.6.

Listing 4.6 – The data transmission between all devices

```
foreach device_A
    load(data_region_A) from device_A
    foreach device_B in neighbor_list_A
        load(ghost_region_B) from device_B
        transferData(region_aTob)
        wait();
```

OpenCL does not assume that data can be transferred directly between devices, so commands only exist to move from a host to a device, or from a device to host. Copying data from one device to another requires an intermediate transfer to the host. Figure 4.9 presents the procedure of data transmission between devices.

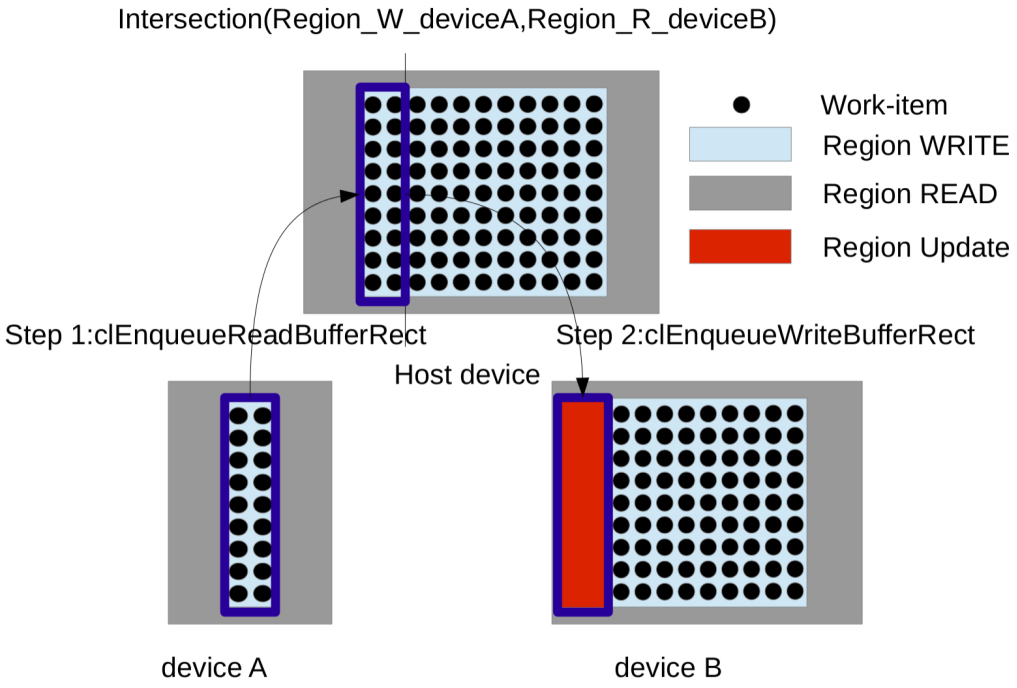


Figure 4.9 – Data transmission between two devices

The pointer of host memory cannot be simultaneously possessed by several devices, one device cannot communicate with host until the data transmissions of other devices have finished.

Listing 4.7 – Resulting READ and WRITE array regions of the 1D-stencil kernel.

```
__kernel void generic_stencil1D(__global float *A, __global float *B) {
    const unsigned int i = get_global_id(0)+1;
    //  <A[PHI1]-R-EXACT-{PHI1==i, i==1}>
    //  <A[PHI1]-R-EXACT-{PHI1==i+1, i==1}>
    //  <A[PHI1]-R-EXACT-{PHI1==i-1, i==1}>
    //  <B[PHI1]-W-EXACT-{PHI1==i, i==1}>
    B[i]= i + (A[i-1]+A[i]+A[i+1])/3;
  }
```

4.6 Generation of Host code

This Section introduces the overall process of host code generation performed by STEPOCL. Based on the kernel(s) and on the configuration file, STEPOCL generates a multi-device OpenCL program. After the initialization of the OpenCL environment, the generated program contains three main components: the *detection* component, the *deployment* component, and the *consistency* component. The generated code ends by collecting and aggregating the result from each device. Algorithm 1 illustrates this process.

Detection component During the initialization (line 1 of Algorithm 1), the generated code detects the available OpenCL devices and associates a ratio to each device thanks to the offline profiler (see Section 4.2). It indicates the percentage of the workload that the generated code has to deploy to each device. After the initialization, the `Devices` variable thus contains a list of `(device, ratio)` pairs.

Deployment component The deployment component (lines 2 to 5 and lines 8 to 12 in Algorithm 1) performs the initial deployment of the kernels on the devices.

First, based on the ratio (provided by the detection component) and on the tile size (provided by the configuration file, see Section 3.2.2) of each device, the generated code computes the number of tasks for each device. For a given device, the number of tasks is simply equal to the size of the output argument multiplied by the ratio and divided by the size of a tile.

Then, with the `data_split` axis provided by the configuration file (see Subsection 3.2.2) and on the list of tasks computed at the previous line, the generated code computes the subset of the input data accessed by each device (line 4 of Algorithm 1). This computation relies on the PIPS compiler. PIPS analyzes the instructions and represents memory accesses as convex polyhedra.As an illustration, Listing 4.7 reports the access patterns to arrays A and B in the 1D-stencil: the kernel reads A at three different positions (i, $i+1$ and $i-1$) while it writes B at the position i. From this information, the generated code computes the convex hull of the data accessed by the tasks of each device, and stores this result in the `Subdata` variable. Due to region representation of PIPS, the size of `Subdata` can be overestimated.

Finally, the generated program copies the data to the devices (lines 8 to 9 of Algorithm 1) and deploys the tasks (lines 11 to 12).

Consistency component This component ensures data consistency and tries to minimize the data transfers between the devices between two iterations. It includes the lines

from 6 to 7 and line 14 of Algorithm 1.

The generated code first identifies the regions that have to be exchanged between the iterations (line 6 of Algorithm 1). Again, the generated code uses the PIPS analysis. As presented in Section 4.5, for each argument exchanged between two iterations, the generated code identifies which regions of the argument are replicated between at least two devices, accessed in read in the input argument and accessed in write in the output argument. For each device, the `FindDeviceNeighboring` function at line 6 identifies both neighbors and their associated replicated regions.

At the end of an iteration (line 15 of Algorithm 1), STEPOCL retrieves the computing results from each devices.

Algorithm 1: Generated host code.

Input:
Data_host: Data location on the host
Data_size: Data size
Data_split: Data-splitting plan
Kernel_tile: Computation size done by each kernel instance
N_iter : Number of iterations

Data:
Devices: List of detected computing devices
$Subtasks_i$: Workload assigned to device i
$Subdata_i$: Data chunk to distribute on device i
Kernel_accesses: Data access pattern by the kernel
$Neighbors_i$: Data neighboring of device i
$Data\_device_i$: Data location on device i

```
1  Devices ← InitOpenCLEnvironment();
2  Subtasks ← PartitionWorkload(Data_size,
3  Kernel_tile, Devices);
4  Subdata ← PartitionData(Data_host, Data_size,
5  Data_split, Subtasks);
6  Neighbors ← FindDeviceNeighboring(Subdata,
7  Kernel_accesses);
8  foreach dev_i in Devices do
9      Data_device_i ← InitDeviceData(dev_i, Subdata_i);
10 for k=1 to N_iter do
11     foreach dev_i in Devices do
12         InvokeKernel(Subtasks_i, Data_device_i);
13     WaitForTermination();
14     UpdateSubData(Subdata, Neighbors);
15 foreach dev_i in Devices do
16     ReadBack(Data_host, dev_i, Data_device_i);
17 FinalizeOpenCLEnvironment();
```

4.7 Conclusion

This chapter introduces the core modules that are implemented in STEPOCL. STEPOCL uses the result of region analysis and information from offline profiling to perform the workload partition. Then, STEPOCL runtime manages the process of workload balancing and the communications between multiple devices.

STEPOCL is well suited to massive data-parallel applications. The data size and cost of communication can greatly affect the performance. The ideal situation is that the data size is large enough to saturate the computing power of the whole platform. As for the

applications with smaller data size, the cost of communication between devices becomes more important, and the better performance might be achieved by using fewer computing devices. In the next chapter, we will evaluate STEPOCL with three applications. For each application, we will analyse the impact of data size and cost of communication on performance.

Chapter 5

Evaluation

Le génie commence les beaux ouvrages, mais le travail les achève.

—*Joseph Joubert*

Contents

We evaluate STEPOCL on three test cases: a 5-point 2D-stencil, a matrix multiplication and an N-body application.

We start by describing the target applications context before evaluating the generated codes according to different criteria:

- we compare the size of handwritten applications to the size of the equivalent applications generated by STEPOCL;
- we evaluate the accuracy of the offline profiler;
- we compare the performance of the handwritten applications to the performance of the equivalent applications generated by STEPOCL when running on a single device;

- we evaluate the performance of the generated applications when running on multiple devices.

5.1 Test cases

In order to assess the STEPOCL usability and performance, we use three applications (Stencil computation, Matrix multiplication and N-body) that can run on heterogeneous multi-device platforms.

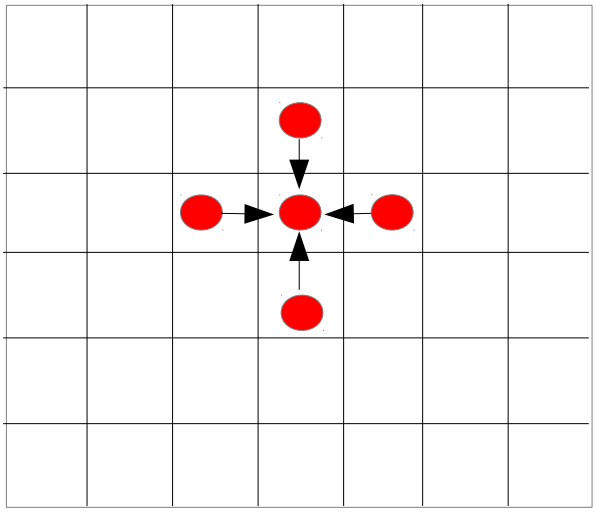


Figure 5.1 – 5-point 2D-stencil computation.

Stencil computation The first application is a 5-point 2D-stencil computation. It takes a 2D matrix of scalar values as input and outputs, for each element, a weighted average of its 4-neighborhood. Figure 5.1 illustrates this computation. We compare the generated stencil application with an handwritten OpenCL version. We set the configuration file (Listing 5.2) so that:

- It defines two kernel arguments: an *input* and an *output* matrices of float elements (line 3 – line 25 of Listing 5.2), which are marked to be split by lines (line 29 – line 32 of Listing 5.2).
- It defines two kernel versions: the first kernel (reported in Listing 5.1) is designed to fit CPU and Xeon Phi devices by setting work-items to work over 4-elements wide tiles while the second kernel is optimized for GPUs and takes advantage of their shared memory. Each work-group instantiates 16×4 work-items that first copy their data to their local shared memory before entering the computation phase and processing 4 elements. (line 34 – line 64 of Listing 5.2)
- The target application iterates ten times, switching input and output matrices after each iteration (line 66 – line 73 of Listing 5.2). The STEPOCL code updates data in device memory in a fixed pattern :when a sub-matrix, corresponding to

the bounded interval $[(x1, y1), (x2, y2)]$, is assigned to a device, the frontiers of its corresponding output are fetched from all its neighbor devices.

Listing 5.1 – 2D-stencil kernel with a tile of 4 elements.

```
__kernel void stencil(__global float *B,
                      __global float *A,
                      unsigned int line_size) {
  const unsigned int x = get_global_id(0);
  const unsigned int y = get_global_id(1);
  A += line_size + 1; // OFFSET
  B += line_size + 1; // OFFSET
  for(unsigned int k=0; k<4; k++){
    B[(y*4 + k)*line_size + x]
    = 0.75 * A[y*4 + k][x]
      + 0.25 * (tile[y*4 + k+1][x-1]
              + tile[y*4 + k+1][x+1]
              + tile[y*4 + k-1][x-1]
          + tile[y*4 + k-1][x+1] );
  }
}
```

List 5.2 presents the configuration file of 2D-stencil application.

Listing 5.2 – Configuration file of 2D-stencil application

```
<?xml version="1.0"?>
<root>
  <argument>
    <ID>B</ID>
    <data_type>float</data_type>
    <arg_size>
      <dim_size axis="x"> 1026 </dim_size> <!--<dim_size>1026</dim_size> -->
      <dim_size axis="y"> 1026 </dim_size> <!--<dim_size>1026</dim_size> -->
    </arg_size>
  </argument>
  <argument>
    <ID>A</ID>
    <data_type>float</data_type>
    <arg_size>
      <dim_size axis="x"> 1026 </dim_size>
      <dim_size axis="y"> 1026 </dim_size>
    </arg_size>
  </argument>
  <argument>
    <ID>line_size</ID>
    <data_type>unsigned int</data_type>
    <depend>
      <axis ID="A">x</axis>
    </depend>
  </argument>
  <kernel>
    <name>stencil</name>
    <target_arg>B</target_arg>
    <data_split>
      <axis ID="B">y</axis>
      <axis ID="A">y</axis>
    </data_split>
    <!--multiple platform is ok-->
    <implem>
      <platform>NVIDIA CUDA</platform>
      <device_type>GPU</device_type>
      <filename>stencilGPU.cl</filename>
      <tile>
        <size axis="x">1</size>
        <size axis="y">4</size>
      </tile>
      <work_group>
```

```
43          <size axis="x">16</size>
44          <size axis="y">4</size>
45        </work_group>
46      </implem>
47      <implem>
48        <platform>Intel(R) OpenCL</platform>
49        <device_type>CPU</device_type>
50        <filename>stencilCPU.cl</filename>
51        <tile>
52          <size axis="x">1</size>
53          <size axis="y">4</size>
54        </tile>
55      </implem>
56      <implem>
57        <platform>Intel(R) OpenCL</platform>
58        <device_type>ACCELERATOR</device_type>
59        <filename>stencilMIC.cl</filename>
60        <tile>
61          <size axis="x">1</size>
62          <size axis="y">4</size>
63        </tile>
64      </implem>
65    </kernel>
66    <control>
67      <loop iterations="10" >
68        <exec>Stencil2D</exec>
69        <switch>
70          <arg ID="A">B</arg>
71        </switch>
72      </loop>
73    </control>
74  </root>
```

Matrix multiplication The matrix multiplication application computes the $C = A \times B$ operation on 2D matrices. We compare the STEPOCL matrix multiplication with the one provided by AMD APP SDK benchmark. We set the related configuration file so that:

- It defines three kernel arguments: the three matrices of float elements A, B and C (line 3 – line 26 of Listing 5.3). C and A are marked to be split by lines (line 30 – line 33 of Listing 5.3), and B is totally replicated on all the devices because of the data dependency analysis.
- It defines two kernel versions (these two kernels are directly taken from AMD SDK): the first one targets CPU and Xeon Phi devices using a tile of 4×4 elements, while the one designed for GPUs also uses a tile of 4×4, but defines work-groups of 4×4 work-items in order to work on shared local memory (line 34 – line 64 of Listing 5.3).
- The generated application executes only one iteration without triggering any communication between devices.

List 5.3 presents the complete configuration file of matrix multiplication.

Listing 5.3 – Configuration file of Matrix multiplication

```
1  <?xml version="1.0"?>
2  <root>
3    <argument>
4      <ID>matA</ID>
```

```
5       <data_type>float</data_type>
6       <arg_size>
7         <dim_size axis="x">1024</dim_size>
8         <dim_size axis="y">1024</dim_size>
9       </arg_size>
10    </argument>
11    <argument>
12      <ID>matB</ID>
13      <data_type>float</data_type>
14      <arg_size>
15        <dim_size axis="x">1024</dim_size>
16        <dim_size axis="y">1024</dim_size>
17      </arg_size>
18    </argument>
19    <argument>
20      <ID>matC</ID>
21      <data_type>float</data_type>
22      <arg_size>
23        <dim_size axis="x">1024</dim_size>
24        <dim_size axis="y">1024</dim_size>
25      </arg_size>
26    </argument>
27    <kernel>
28      <name>floatMatrixMult</name>
29      <target_arg>matC</target_arg>
30      <data_split>
31        <axis ID="matA">y</axis>
32        <axis ID="matC">y</axis>
33      </data_split>
34      <implem>
35        <platform>NVIDIA CUDA</platform>
36        <device_type>GPU</device_type>
37        <filename>multGPU.cl</filename>
38        <tile>
39          <size axis="x">4</size>
40          <size axis="y">4</size>
41        </tile>
42        <work_group>
43          <size axis="x">4</size>
44          <size axis="y">4</size>
45        </work_group>
46      </implem>
47      <implem>
48        <platform>Intel(R) OpenCL</platform>
49        <device_type>CPU</device_type>
50        <filename>mult.cl</filename>
51        <tile>
52          <size axis="x">4</size>
53          <size axis="y">4</size>
54        </tile>
55      </implem>
56      <implem>
57        <platform>Intel(R) OpenCL</platform>
58        <device_type>ACCELERATOR</device_type>
59        <filename>mult.cl</filename>
60        <tile>
61          <size axis="x">4</size>
62          <size axis="y">4</size>
63        </tile>
64      </implem>
65    </kernel>
66  </root>
```

N-body The N-body application simulates the collective motions of a large particle set under Newtonian forces in a 3D space. At each iteration, each particle, defined by a mass and a velocity, changes both its position and its velocity by using the position and the

velocity of all the other particles. We compare the generated N-body application with the one provided by AMD APP SDK benchmark.

In STEPOCL, we define the following arguments (line 3 – line 42 of Listing 5.4):

- Three integer: the number of particles, a softening factor and an elapsed time interval. All these arguments are replicated because of the data dependency analysis;
- Two arrays of float elements containing the particles positions: one stores the positions at the previous iteration while the other, the current ones;
- Two arrays of float elements that contain the velocity of each particle: one array stores the velocities at the previous iteration while the other contains the current ones.

The application performs ten iterations. After each iteration, it switches the input and output arrays (line 75 – line 82 of Listing 5.4). Considering the communication, the two input arrays are replicated on all the devices because of the data dependency analysis. Indeed, to compute the new position and a new velocity of one particle, all particle positions and velocities of the previous iteration are used. As the input and output arrays are switched after an iteration, between each iteration, each device broadcasts its output to all the other devices.

List 5.4 presents the complete configuration file of N-body.

Listing 5.4 – Configuration file of N-body

```
1  <?xml version="1.0"?>
2  <root>
3    <argument>
4      <ID>Pos</ID>
5      <data_type>float</data_type>
6      <arg_size>
7        <dim_size axis="x">4096</dim_size>
8      </arg_size>
9    </argument>
10   <argument>
11     <ID>Vel</ID>
12     <data_type>float</data_type>
13     <arg_size>
14       <dim_size axis="x">4096</dim_size>
15     </arg_size>
16   </argument>
17   <argument>
18     <ID>numParticles</ID>
19     <data_type>int</data_type>
20   </argument>
21   <argument>
22     <ID>delT</ID>
23     <data_type>float</data_type>
24   </argument>
25   <argument>
26     <ID>espSqr</ID>
27     <data_type>float</data_type>
28   </argument>
29   <argument>
30     <ID>newPosition</ID>
31     <data_type>float</data_type>
32     <arg_size>
33       <dim_size axis="x">4096</dim_size>
34     </arg_size>
35   </argument>
36   <argument>
```

```
37      <ID>newVelocity</ID>
38      <data_type>float</data_type>
39      <arg_size>
40        <dim_size axis="x">4096</dim_size>
41      </arg_size>
42    </argument>
43    <kernel>
44      <name>nbody_sim</name>
45      <target_arg>Pos</target_arg>
46      <data_split>
47        <axis ID="Pos">x</axis>
48        <axis ID="Vel">x</axis>
49        <axis ID="newPosition">x</axis>
50        <axis ID="newVelocity">x</axis>
51      </data_split>
52      <implem>
53      <platform>NVIDIA CUDA</platform>
54      <device_type>GPU</device_type>
55      <filename>nbodyGPU.cl</filename>
56      <tile>
57        <size axis="x">4</size>
58      </tile>
59      <work_group>
60        <size axis="x">128</size>
61      </work_group>
62      </implem>
63      <implem>
64        <platform>Intel(R) OpenCL</platform>
65        <device_type>ALL</device_type>
66        <filename>nbody.cl</filename>
67        <tile>
68          <size axis="x">4</size>
69        </tile>
70        <work_group>
71          <size axis="x">128</size>
72        </work_group>
73      </implem>
74    </kernel>
75    <control>
76      <loop iterations="10">
77        <switch>
78          <arg ID="Pos">newPosition</arg>
79          <arg ID="Vel">newVelocity</arg>
80        </switch>
81      </loop>
82    </control>
83  </root>
```

5.2 Volume of the generated source code

Table 5.2 reports the number of lines of code of the three native OpenCL applications as well as the size of the three configuration files that STEPOCL used for generating the tested applications. The table also contains the number of lines of code of the applications generated by STEPOCL. The kernels consist of a few tens of lines of code. However, the native OpenCL applications and generated host codes in charge of instantiating the kernels on the devices consist of several hundreds lines of code.

The native OpenCL applications are only able to run on single device, while the STEPOCL applications can run on multiple devices. We classify the generated code between what is part of the STEPOCL runtime and what is part of the application. The runtime source code is generic and does not vary from one application to another, while the application code is more specific and may vary depending on the description

Test case	Native OpenCL application (kernels included)	Kernels	STEPOCL config. file	STEPOCL generated code (kernels included)
Stencil	490	GPU=51/others=18	74	1153
Mat. Mult.	1212	GPU=103/others=78	66	1216
N-body	1041	81	83	1116

Table 5.1 – Generated STEPOCL code size (in lines).

	# of lines of code	
	Application	Runtime
Initialization	89 loc	323 loc
Workload partitioning	224 loc	76 loc
Communication	24 loc	322 loc
Retrieve results	85 loc	10 loc
Total	422	731

Table 5.2 – Distribution of the lines of code of the generated 2D stencil application

file provided by the user or on the OpenCL kernel. The code is also classified depending on its semantic: the code may be in one of the 4 parts of the generated host code as presented in Section 3.2.3: the *initialization*, the *workload partitioning*, the *communication management* or the *retrieving of the results*. Table 5.2 reports the analysis of generated 2D stencil source code. Most of the lines of code are dedicated to the runtime system. The generated application itself remains compact (422 lines of code), but may increase for more complex computational kernels.

The *workload partitioning*, which represents the largest part of the generated code, includes the initial distribution of the workload across devices as well as the load balancing mechanism. This part of the code is more dependant on the type of application. Thus, most of the source code is located in the application. *Retrieving the results* is also mostly performed by the application and the runtime system only provides a function for transferring data back from the devices.

The other parts of the code are more generic and mostly provided by the runtime. The *initialization* mainly consists in detecting the available devices and initializing the OpenCL environment, while the generated code simply declares variables and calls STEPOCL runtime functions. The *communication management*, which is in charge of synchronizing the devices and managing the data exchanges between devices, is mainly implemented by the runtime. The STEPOCL runtime implements transfer primitives as well as functions that perform the polyhedral analysis.

To conclude, we can observe that the STEPOCL configuration files contains only few tens of lines of code, which is roughly ten time smaller than the generated code. This result shows that STEPOCL simplifies the development of an application for multiple device.

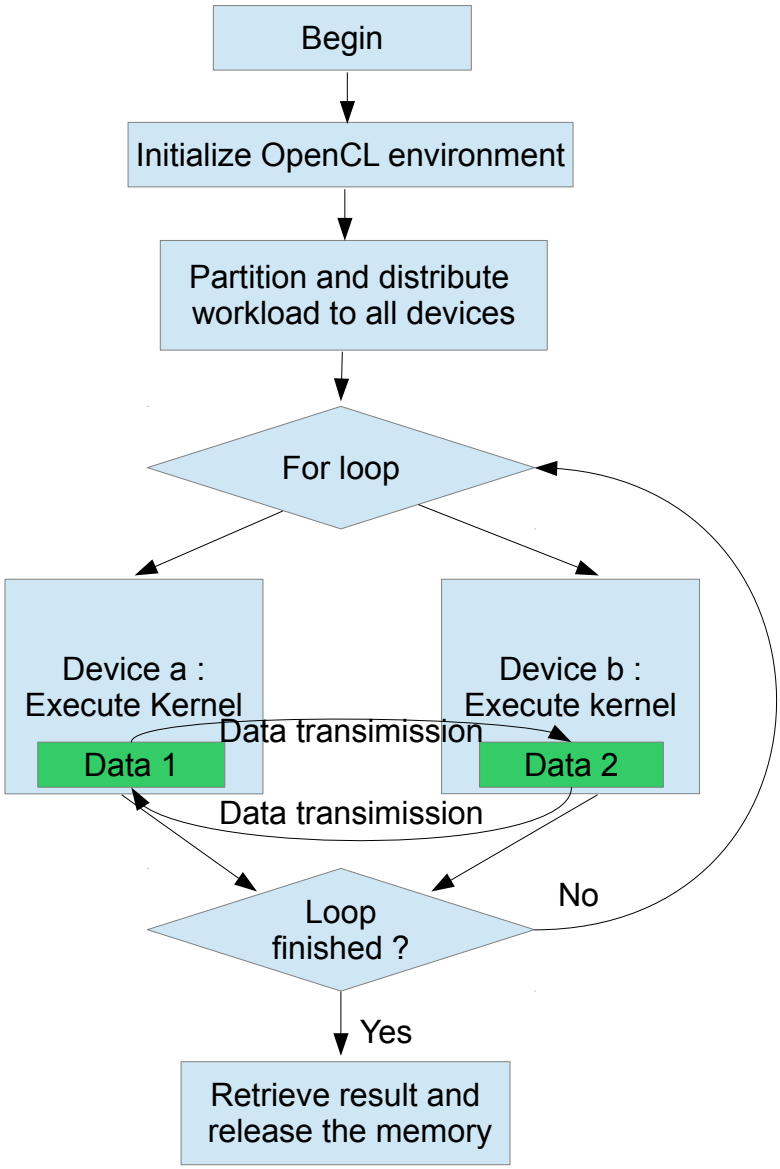


Figure 5.2 – the structure of generated 2D stencil code

5.3 Performance evaluation

We now evaluate the performance of the three applications on two heterogeneous platforms.

5.3.1 Experimental platforms

We summarize the characteristics of the two platforms in Table 5.3: the HANNIBAL platform is a dual quad-core Intel Xeon CPU with three Quadro FX 5800 GPUs while

Name	Hannibal	Mistral
CPU models	2 x Intel Xeon X5550	2 x Intel Xeon E5-2670
# CPU cores	2 x 4	2 x 10
# threads	2 x 8	2 x 10
CPU frequency	2.66 GHz	2.50GHz
OpenCL support	Intel OpenCL 1.1	Intel OpenCL 1.2
Accelerator type	NVIDIA GPU	Intel Xeon Phi
Models	3 x Quadro FX 5800	2 x Xeon Phi 7120P
# cores	3 x 240 CUDA cores	2 x 61 cores (2 x 244 threads)
Processor clock	1296 MHz	1238MHz
Memory	3 x 4096 MB GDDR3	2 x 16 GB GDDR5
OpenCL support	NVidia OpenCL 1.1	Intel OpenCL 1.2

Table 5.3 – Experimental platform outline.

Mistral is a dual 10-cores Intel Xeon CPU with two Xeon Phi accelerators.

5.3.2 Evaluation of the profiler

Before evaluating the performance of the applications, we evaluate the offline profiler by measuring how it converges towards a balanced distribution of the workload.

Figure 5.3 depicts the workload distribution w_i^j and the measured duration d_i^j of the CPU and GPU devices on a machine running a 7-point 3D-stencil. Artificially, for the first iteration of the application, we assign 99 % of the workload to the GPU and the remaining 1 % to the CPU. After the first iteration, the profiler detects that it assigned too much workload to the GPU. The workload partition for the second iteration thus assigns more work-items to the CPU, which results in a more balanced execution time. Yet, the difference between d_0^{t2} and d_1^{t2} leads to assigning more work-items to the CPU for the third iteration. After the third iteration, the profiler detects that the difference between d_0^{t3} and d_1^{t3} is small enough and stops the calibration process. In this experiment, we run a few more iterations in order to make sure that the execution on each device is stable when the workload distribution does not change.

5.3.3 Comparison with the reference codes

In order to ensure that the code generated by STEPOCL does not degrade the performance as compared to a native implementation written directly in OpenCL, we compare the performance of the generated application with the native OpenCL applications.

As the native OpenCL applications are only designed to run on a single device, we only use a single GPU on Hannibal. From Table 5.4, we can observe that STEPOCL does not change the performance of the matrix multiplication and that STEPOCL introduces an overhead of 1% for stencil and 2% for N-body. From this result, we can thus conclude that STEPOCL does not significantly modify the performance on a single accelerator.

5.3.4 Stencil

Figure 5.4 presents the performance of the generated stencil code on the two machines. Our measures underlines the performance scalability achieved by STEPOCL, with GPU and CPU devices adding up their computational horsepower efficiency.

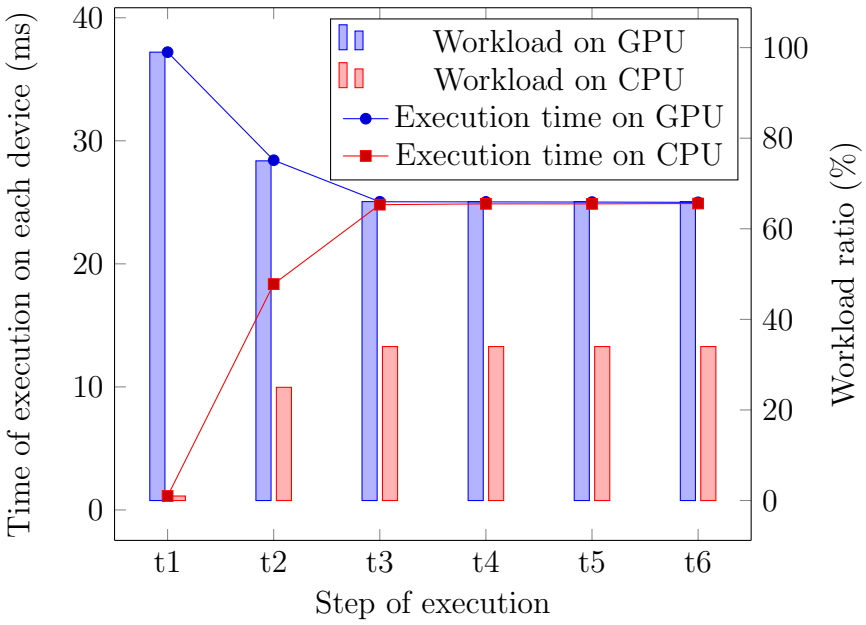


Figure 5.3 – Workload adjustment performance of the 3D-stencil application.

On Hannibal (see Figure 5.4(a)), we observe that the peak performance achieved when using all the available devices (91.8 GFLOPS) roughly corresponds to the sum of the performance of each device running individually (14.5 GFLOPS for 1 CPU, 28.1 GFLOPS for 1 GPU). The efficiency is not 100 % because of the communication between the devices that are required when computing the stencil on multiple devices in order to ensure data consistency.

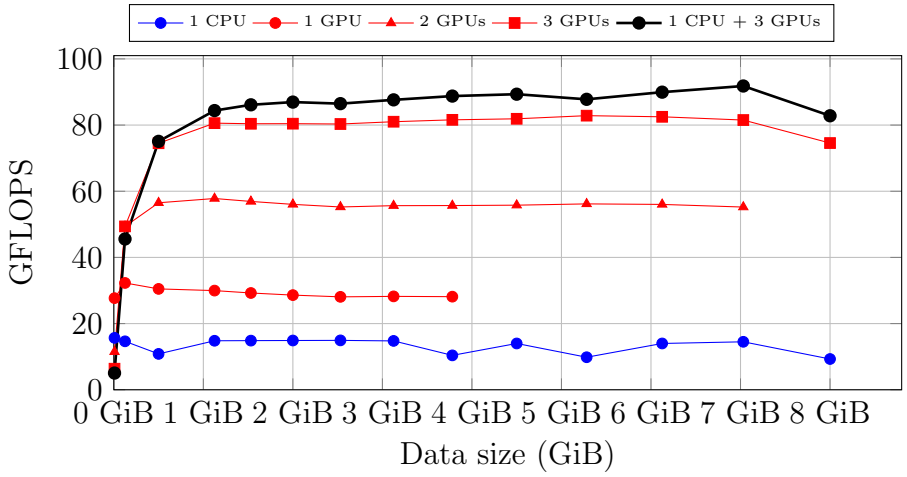
The results on the mistral platform have a similar trend: the peak performance when using all the devices (134.1 GFLOPS) approximately corresponds to the sum of the individual performance (20.6 GFLOPS on 1 CPU, 63.1 GFLOPS on 1 Xeon Phi).

From these two results, we can conclude that STEPOCL scales linearly with the small number of devices. This result also confirms that the offline profiler seems to provide efficient ratios able to perfectly balance the load between the devices.

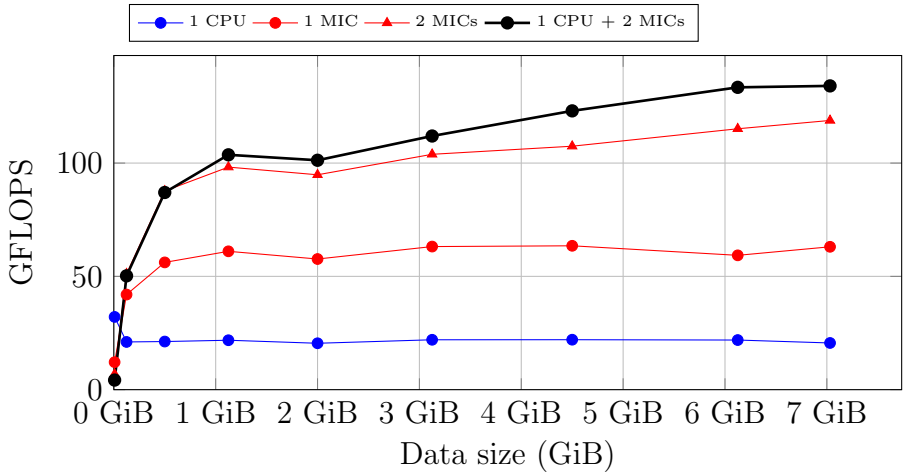
Moreover, STEPOCL pushes forward the memory limits by automatically distributing data sets which are too large to be processed by a single device, summing up the memory of multiple devices to handle larger problems. On the Hannibal platform, where GPUs are equipped with 4 GiB of memory, the 1-GPU version cannot process the test cases that require more than 4 GiB of memory. Similarly, the 2-GPUs version is limited to 8 GiB, while the versions that exploit the 3-GPUs can process larger problems.

Application name	2D-Stencil	Mat. Mult.	N-body
Workload	4096x4096	1024x1024	32768 particles
Relative performance	0.99	1.00	0.98

Table 5.4 – Relative performance of STEPOCL as compared to a native OpenCL implementation on Hannibal.



(a) Performance on HANNIBAL

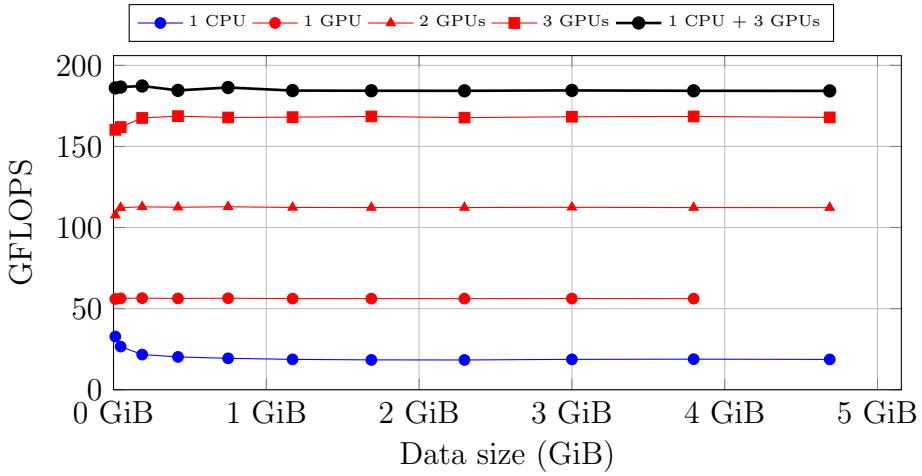


(b) Performance on MISTRAL

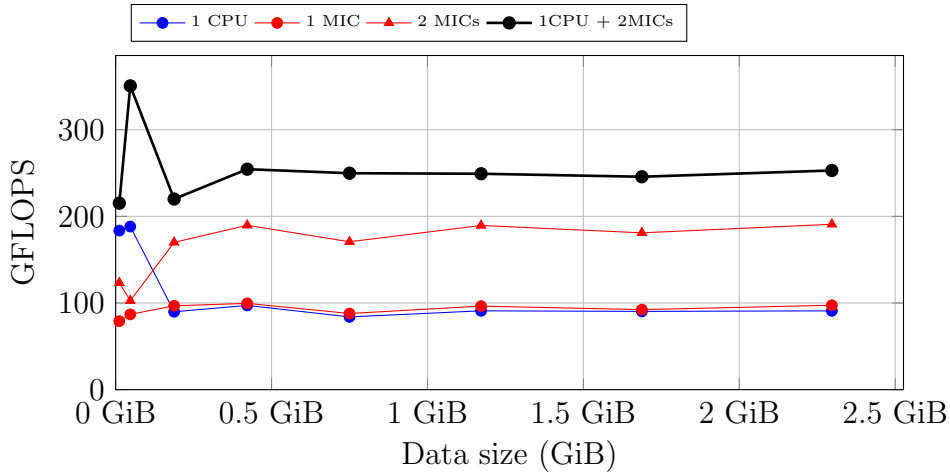
Figure 5.4 – Performance of the 5-point 2D-stencil application. The horizontal axis corresponds to the size the input and output matrices required to solve the problem.

5.3.5 Matrix multiplication

Figure 5.5 presents the performance of the generated matrix multiplication on the HANNIBAL and MISTRAL machines. On HANNIBAL (see Figure 5.5(a)), the performance achieved when using both the CPU and the 3 GPUs (184.3 GFLOPS) almost corresponds to the sum of the performance achieved by each device individually (18.36 GFLOPS on the CPU, 56.16 GFLOPS on each GPU). On MISTRAL (see Figure 5.5(b)), when using the CPU and the 2 Xeon Phis (252.9 GFLOPS), the performance corresponds to 88 % of the one achieved by each device individually (90.9 GFLOPS on the CPU, 97.4 GFLOPS on each Xeon Phi). Thus, in term of performance, the STEPOCL generated code scales up correctly. Moreover, as for the stencil application, the 1-GPU version is bounded by its inner memory size. Once again, STEPOCL allows to compute bigger problem than the original code thanks to its multi-device dimension.



(a) Performance on Hannibal



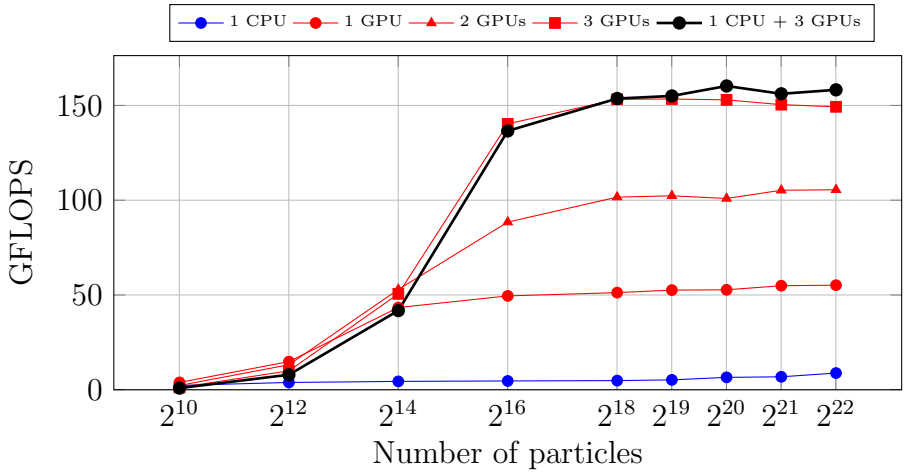
(b) Performance on Mistral

Figure 5.5 – Performance of the matrix multiplication application. The horizontal axis corresponds to the summed size of the A, B, and C matrices.

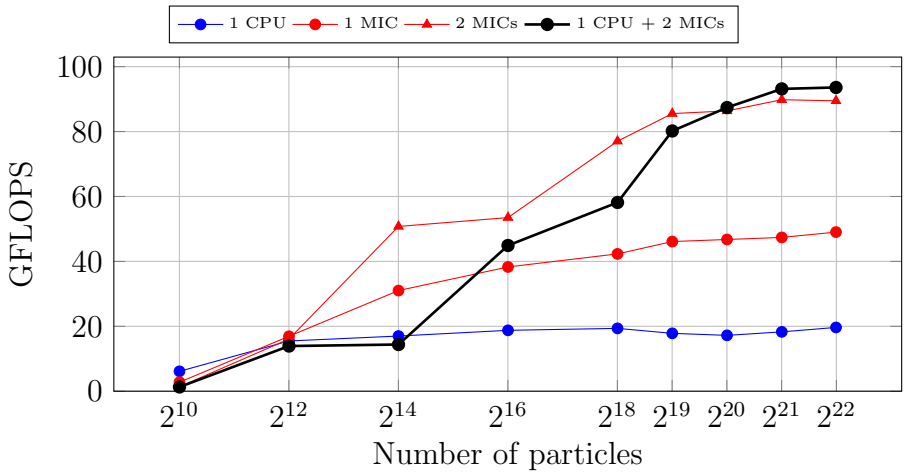
5.3.6 N-body

Figure 5.6 presents the performance of the generated N-body application on Hannibal and Mistral. On Hannibal (see Figure 5.6(a)), the performance achieved when using both the CPU and the 3 GPUs (152.8 GFLOPS) corresponds to 94 % of the cumulated performance achieved by each device individually (9.38 GFLOPS on the CPU, 51.12 GFLOPS on each GPU). On Mistral (see Figure 5.6(b)), when using the CPU and the 2 Xeon Phis (90 GFLOPS), the performance corresponds to 83 % of the one achieved by each device individually (16.31 GFLOPS on the CPU, 45.68 GFLOPS on each Xeon Phi).

As described in Subsection 5.1, the updated data communication between each iteration is highly critical in comparison with our two other test cases: after each iteration, the computed data is broadcasted to all devices attending the computation. This is why



(a) Performance on HANNIBAL



(b) Performance on MISTRAL

Figure 5.6 – Performance of the N-body application.

the generated N-body code does not scale well on small data sets, particularly on Xeon Phi as cross-device communication are very expensive on this platform. However, the computation time increases in quadratic time with respect to the workload, while the communication time only increases linearly. For this reason, with larger workloads, the N-body application scales linearly with the number of devices.

The N-body application does not deal with memory footprint as all arrays need to be duplicated. Since computation grows quadratically, it really focuses on the reduction of the makespan. STEPOCL reaches this objective with its multi-device dimension.

5.4 Analysis of overhead of communication

The time of communication is decided by the bandwidth of network and the size data which is needed to be transferred. The bandwidth of network is fixed, however the

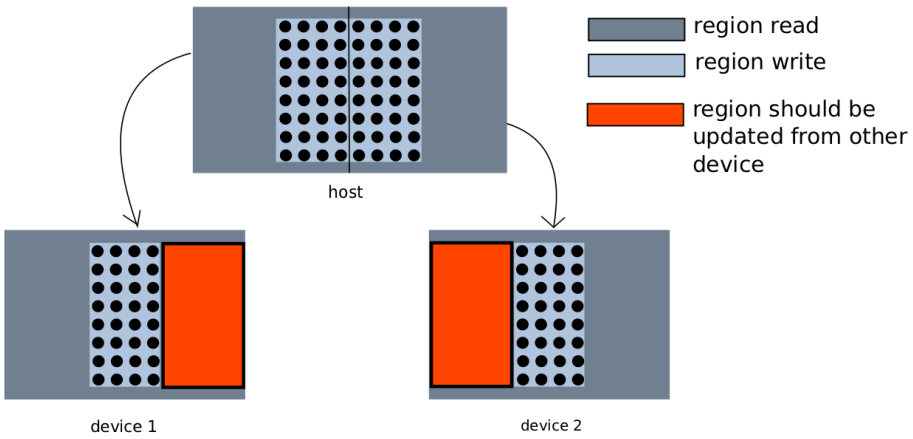


Figure 5.7 – Partitioning data by column

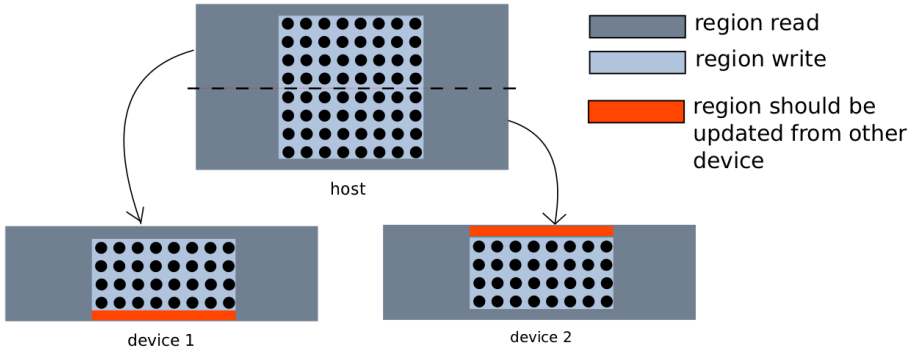


Figure 5.8 – Partitioning data by row

transferring data size can be significantly affected by the partition layout.

Listing 5.5 – Stencil Loop Example

```
for(int t=0; t<T; ++t){
  for(int i=10;i<N-10;++i){
    for(int j=10;j<N-10;++j){
      A[i][j]=CNST * (B[i][j+10]+B[i][j-10]
          +B[i-1][j]+B[i+1][j]);
    }
  }
  swap(A,B);
}
```

For example, the Listing 5.5 presents a part of typical stencil loop code. In this case, updating the matrix A depends more on the data which are relatively allocated on X axes. If we partition the data space by row (Figure 5.8) instead of partitioning by column (Figure 5.7), we can reduce significantly the overhead of communication.

Another factor that affects the overhead of communication is the continuity of transferred region. STEPOCL use *clEnqueueReadBufferRect* and *clEnqueueWriteBufferRect* to transfer a 2D or 3D region between a buffer object and host memory, however the physical address of a 2D or 3D region is not always continuous. In Figure 5.9, splitting

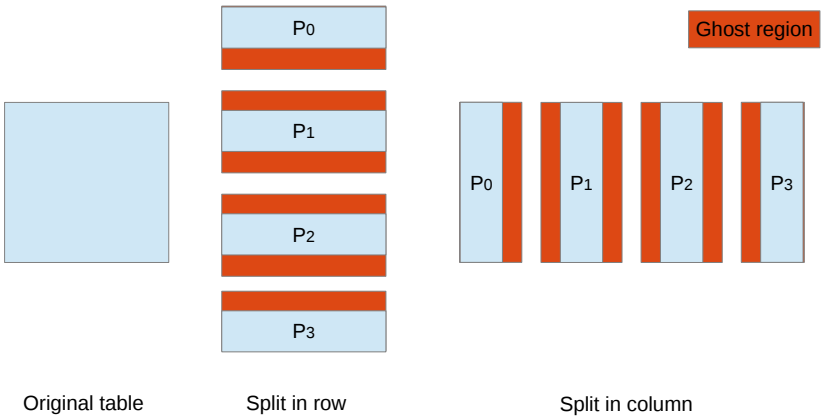


Figure 5.9 – Data partition of a 2D table

table in row can keep the continuity of physical addresses for each ghost region and splitting in column will break this continuity. According to our experiment, transferring a continuous region is much more efficient than transferring an uncontinuous region. Thus, in the situation of Figure 5.9, splitting in line is a better option.

Lastly, the overhead of communication also could be caused by repeatedly adjustment of workload proportion for each OpenCL devices. For the applications which require consistently to relaunch computing kernels (such as stencil iteration and N-body), STEPOCL will observe the execution time on each device and adjust the workload proportion when the workload is obviously unbalanced between devices (see section 4.4). Once the adjustment happens, STEPOCL aggregates all computed data from each devices and redistributes it in adjusted proportion. Thus once a re-adjustment of workload happens, the size of input data and the bandwidth of network will decide the duration of readjustment. Fortunately, STEPOCL profiler and the constraint implemented in runtime ensure that the readjustment will not happen too frequently.

5.5 Conclusion

In this chapter, we have presented the usage of STEPOCL and the performance of generated code on heterogeneous architectures. According to the results of evaluation, by using STEPOCL, the number of lines of code to write an application to an application for multiple devices is drastically reduced. STEPOCL generates more than one thousand lines of code from a configuration file less than one hundred lines of code. The generated applications do not degrade the performance as compared to a native implementation written directly in OpenCL. Thanks to STEPOCL, the mechanisms of workload partitioning and workload balancing ensure that the generated applications can exploit the full potential of computing power of any multiple devices heterogeneous architectures, the performance of generated applications scales linearly with the number of devices.

Chapter 6

Conclusion

Choisissez un travail que vous aimez, et vous n'aurez jamais à travailler un seul jour dans votre vie.

—*Confucius*

Contents

Heterogeneous architectures are being wildly used in the domain of High performance computing. However, due to the multiple types of devices with different processing capabilities, programming on these architectures is difficult.

In this thesis, we have introduced STEPOCL, a programming tool that eases the development of applications for multiple heterogeneous devices. Based on an OpenCL compute kernel and a STEPOCL configuration file provided by user, STEPOCL generates an offline profiler to guide the partitioning of the workload and generates the OpenCL host part of the application. The generated application schedules the workload according to the profiling results, launches their execution, and performs the necessary data exchanges between devices.

We evaluated STEPOCL with three applications: a stencil application, a matrix multiplication, and an N-body application. We measured the performance of these applications on two different multi-device platforms. Our evaluation shows that, thanks to STEPOCL, the number of lines of code to write an application for multiple devices is drastically reduced. Our measurements also show that the code generated by STEPOCL can run on complex multi-device systems and that its performance scales well with the number of devices. Using multiple devices also enables to cope with problem sizes that cannot fit into a single accelerator.

6.1 Contribution

The contributions of this thesis are:

1. A programming tool STEPOCL that eases the development of applications for multiple heterogeneous devices.

2. A domain specific language, based on XML, which consists in describing the execution scheme and the data layout processed by OpenCL kernels. Using this language in our context improves the productivity of the developer by decreasing the number of line of codes required to implement an application, but also has the advantage of avoiding many bugs caused by the use of a low-level language such as C for the development.

3. A workload partitioning model which guarantees the balance of workload on multiple devices.

4. A strategy of managing data transmission which maximizes the re-utilization of data transferred in devices memory and minimize the data transmission by only transferring the data that are need for the computation on other devices.

6.2 Limitations

The usability of STEPOCL is limited by the capability of region analysis. STEPOCL uses compiler PIPS as a tool of region analysis. During our test of region analysis, we find that PIPS is not capable of analysing the regions which contain complex logical operations.For example, the result of region analysis may be not correct due to the logical operations presented in Listing 6.1.

Listing 6.1 – Limitation of region analysis

```
1
2   __kernel void func(...)
3   {
4     ... ...
5     {
6       const int cbx = xloc;
7       const int cby = (yloc & 1) ? -1 : 16;
8       const int bx = (yloc & 2) ? cbx : cby;
9       const int by = (yloc & 2) ? cby : cbx;
10    }
11      ... ...
12  }
```

Another limitation is form of distribution. STEPOCL only performs regular data partitioning, such as splitting data in line, in column. The number of sub-data is equal to the number of available devices, thus some distributions, such as distribution in cycles or in diagonal (distribution (**d**) and distribution (**e**) in figure 6.1) are not available in STEPOCL for now.

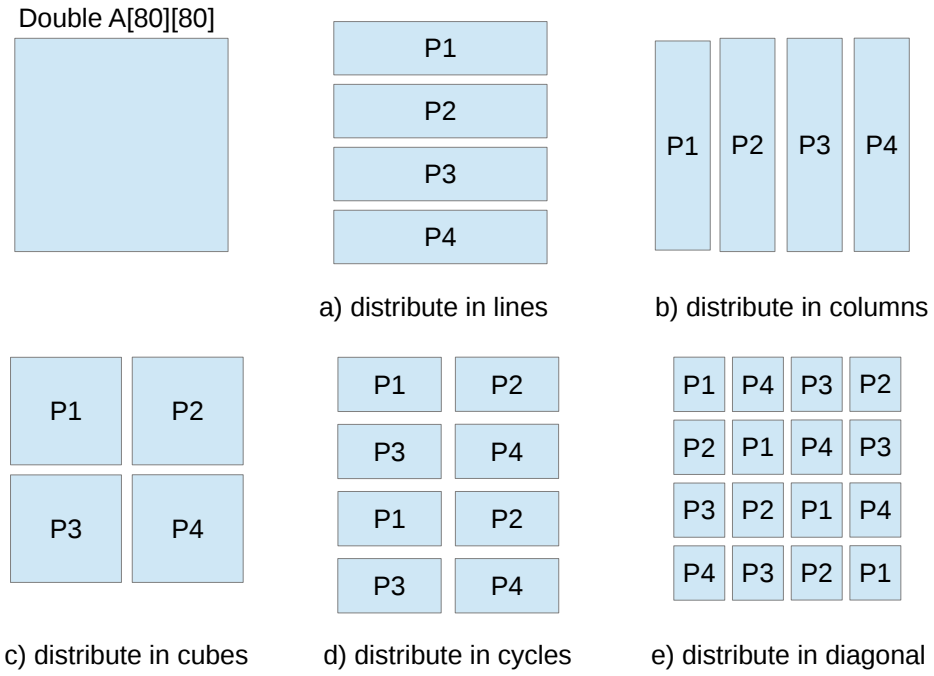


Figure 6.1 – Distributions of a 2D table on 4 devices

6.3 Perspectives

Heterogeneous computing is becoming popular in every domain of computing – from high performance clusters to low-power embedded devices. As a prototype of programming tool, STEPOCL hasn't considered every aspect of programming on heterogeneous architectures. There are several directions we would like to explore.

Perspectives in the short term

Reducing the overhead of data transmission

We would like to further reduce the overhead of data transmission between devices. Currently, STEPOCL only transfers the necessary data which is needed for computing on other devices, then we launch the kernels once all data is updated on every device. We would like to "hide" the overhead of data transmission by overlapping the data transmission with kernel executions.

Optimisation of OpenCL kernel

Although OpenCL applications are portable across different platforms, their performances are not equally portable. In our experience, we achieved the best performance by manually providing device-specific kernels. In the future, we would like to implement an heuristic

module and a code transformation module in STEPOCL so that STEPOCL could automatically optimise computing kernels for specific devices.

Upgrading STEPOCL profiler

Current estimation of the performance on each computing device is based on the hardware statistics, such as the number of cores and their frequency. We would like to develop an heuristics not based on hardware statistics but also based on the static and runtime feature of computing kernel. Thus, the upgraded profiler can provide a more reliable information for workload partitioning.

Perspectives in the long term

Distributed memory architectures

The current version of STEPOCL only works on shared memory architectures. We would like to extend STEPOCL to adapt to distributed memory architectures. Thus, we need to develop a new runtime system which is capable of evaluating the performance of each node on cluster, performing workload partition and scheduling tasks between several nodes. We would like to combine STEPOCL with MPI to synchronize not only the tasks of multiple devices on a single node but also the tasks on different nodes.

Scheduling policies

We would like to implement different scheduling policies to face the challenges of executing more complex tasks on multiple devices heterogeneous architectures. Rather than simply scheduling the tasks to saturate all computing resources, we want to consider more aspects, such as the priorities of tasks and the efficiency of the use of device memories. The options of scheduling policies will be implemented in the STEPOCL interface. Therefore programmer will have more choice for optimising OpenCL applications.

Bibliography

[1] What is CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[2] Bolt Docmentation. http://hsa-libraries.github.io/Bolt/html/.

[3] Boost librairies homepage. http://www.boost.org/.

[4] Ieee standard for information technology-portable operating system interface (posix)-part 1: System application program interface (api)- amendment d: Additional real time extensions [c language], 1999.

[5] TOP500 list of supercomputers, 2015. http://www.top500.org/lists/2015/06/.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[7] Steffen Christgau, Johannes Spazier, Bettina Schnor, Martin Hammitzsch, Andrey Babeyko, and Joachim Waechter. A comparison of cuda and openacc: Accelerating the tsunami simulation easywave. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–5, Feb 2014.

[8] Béatrice Creusillet. *Array region analyses and applications*. PhD thesis, École Nationale Supérieure des Mines de Paris, 1996.

[9] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *Languages and Compilers for Parallel Computing*, volume 1033, pages 46–60. Springer, 1996.

[10] J. Cronsioe, B. Videau, and V. Marangozova-Martin. Boast: Bringing optimization through automatic source-to-source transformations. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 129–134, Sept 2013.

[11] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[12] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 161–172. ACM, 2013.

[13] Rachid HABEL. *High Performance Programming for Hybrid Architectures.* Theses, Ecole Nationale Supérieure des Mines de Paris, November 2014.

[14] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie Christine Counilh, and Raymond Namyst. Toward OpenCL Automatic Multi-Device Support. In *Euro-Par 2014 Parallel Processing*, pages 776–787, 2014.

[15] F. Irigoin. Interprocedural analyses for programming environments. In *In Environments and Tools for Parallel Scienti Computing*, pages 333–350. Elsevier Science Publisher, 1992.

[16] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ICS*, 1991.

[17] Lee Janghaeng, Samadi Mehrzad, Park Yongjun, and Mahlke Scott. Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems. In *Proceedings of PACT '13*, pages 245–256, 2013.

[18] Jim Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming.* Elsevier Waltham (Mass.), Amsterdam, Boston (Mass.), Heidelberg..., et al., 2013.

[19] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. INSPIRE: The insieme parallel intermediate representation. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0:7–17, 2013.

[20] Kim Jungwon, Kim Honggyu, Lee Joo Hwan, and Lee Jaejin. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 277–288. ACM, 2011.

[21] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 149–160. ACM, 2013.

[22] Chris McClanahan and Georgia Tech. History and evolution of gpu architecture.

[23] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. Step: A distributed openmp for coarse-grain parallelism tool. In *OpenMP in a New Era of Parallelism*, volume 5004, chapter 8, pages 83–99. Springer, 2008.

[24] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[25] Faber Peter and Groblinger Armin. A comparison of gpgpu computing frameworks on embedded systems. In *13th IFAC and IEEE Conference on Programmable Devices and Embedded SystemsâĂŤPDES 2015*, volume 48, pages 240–245. ScienceDirect, 2015.

[26] James Reinders and Intel. An overview of programming for intel xeon processors and intel xeon phi coprocessors, 2012.

[27] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi. A comparison of performance tunabilities between opencl and openacc. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 147–152, Sept 2013.

[28] Krishnahari Thouti and S. R. Sathe. Comparison of openmp & opencl parallel processing technologies. *CoRR*, abs/1211.2038, 2012.

[29] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

Appendices

List of publications

[1] Pei Li, Elisabeth Brunet, and Raymond Namyst. High performance code generation for stencil computation on heterogeneous multi-device architectures. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*, pages 1512–1518, 2013.

[2] Pei Li, Elisabeth Brunet, François Trahay, Christian Parrot, Gaël Thomas, and Raymond Namyst. Automatic opencl code generation for multi-device heterogeneous architectures. In *44rd International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*, 2015.